TIPI PRIMITIVI: LIMITI

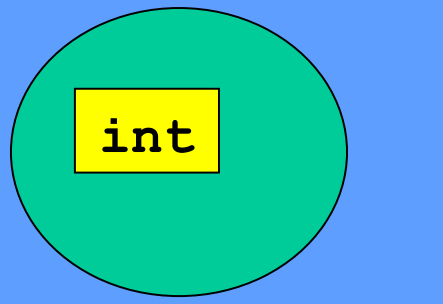
- **I tipi primitivi sono i "mattoni elementari" del linguaggio**
- **In varie situazioni può però essere necessario *trattare i tipi primitivi come oggetti***
  - **quando una funzione pretende come parametro (o restituisce) un `Object`**
  - **quando occorre inserirli in una struttura dati (un array, uno stack, una lista..) che per ipotesi contiene degli `Object`**
- **Poiché una variabile di un tipo primitivo *non è un oggetto*, ciò non è direttamente possibile.**

DAI TIPI PRIMITIVI AGLI OGGETTI

- **Soluzione:** ***costruirsi una classe che incapsuli una variabile di un tipo primitivo***
  - **una classe `MyInt` che incapsuli un `int`**
  - **una classe `MyFloat` che incapsuli un `float`**
  - **...**
- **Ma non ce n'è bisogno: sono già fatte!**
- **Tali classi sono note in Java con l'appellativo di *classi wrapper*** *(classi "avvolgenti")*
- **Ogni classe wrapper ha un *nome (quasi) identico al tipo primitivo che incapsula*, ma con l'iniziale maiuscola.**

TIPI PRIMITIVI E CLASSI “WRAPPER”

TipoprimitivoClasse “wrapper”corrisp	



Ogni classe wrapper definisce metodi per estrarre il valore della variabile incapsulata e viceversa.

TIPI PRIMITIVI E CLASSI “WRAPPER”

- **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*`)`
  - `...`

CLASSI “WRAPPER”: FUNZIONALITÀ

Cosa offre una classe wrapper?

- **un costruttore per incapsulare un valore primitivo. Ad esempio,**

```
Integer wi = new Integer(33);
```

- **un metodo per estrarre il valore incapsulato. Ad esempio,**

```
int k = wi.intValue();
```

- **un metodo toString per rappresentare come stringa il valore incapsulato. Ad esempio:**

```
System.out.println(wi);
```

CLASSI "WRAPPER": FUNZIONALITÀ

Cosa non offre una classe wrapper?

- **operatori aritmetici per fare le operazioni.**
  Ad esempio, *non si può scrivere:*

```
new Integer(3) + new Integer(2);
```

E allora come si fa?

- **si estraggono i valori primitivi incapsulati, …**
- **… si svolgono le operazioni su essi,…**
- **… e infine si costruisce un nuovo oggetto:**

```
Integer wk =
   new Integer(wi.intValue() + wj.intValue());
```

CLASSI WRAPPER - ESEMPIO 1

```
public class EsempioWrapper1 {
 public static void main(String args[]){
   int x = 35;
   Integer wx = new Integer(x);
   Integer wy = new Integer(4);
      // Integer wz = wx + wy;  // NO!
   System.out.println(wx);
   System.out.println(wy);
   Integer wz = new Integer(
   wx.intValue() + wy.intValue() );
   System.out.println(wz);
 }
}
```

```
35
4
```

```
39
```

Si passa nel dominio degli `int` per svolgere l'operazione di somma (che non esiste fra `Integer`)

CLASSI WRAPPER - ESEMPIO 2

```
public class EsempioWrapper2 {
 public static void main(String args[]){
   Integer wx = new Integer(35);
   int x = 2 * wx.intValue();
   System.out.println("wx = " + wx);
   System.out.println("x = " + x);
 }
}
```

```
wx = 35
x = 70
```

La prima stampa usa la **`toString`** di **`Integer`** per ottenere la stringa **`"35"`** e concatenarla alla costante **`"wx = "`**

La seconda stampa usa invece la *conversione automatica* ***`int/String`*** per ottenere la stringa **`"70"`** da concatenare alla costante **`"x = "`**

CLASSI “WRAPPER”… MA NON SOLO

- **La classe wrapper non svolge solo la funzione di "classe avvolgente" per un tipo primitivo**
- **Nella sua parte statica ospita anche *funzioni che operano sul tipo primitivo corrispondente***
  - Si tratta di funzioni (statiche) che *non hanno nulla a che vedere* con la classe wrapper in quanto ADT !
  - Sono "messe lì" semplicemente perché *si doveva pur metterle da qualche parte...* (in Java, non possono esistere funzioni definite "fuori" da una qualche classe)
  - …e la classe wrapper *era il posto più logico* in cui metterle.

CLASSI “WRAPPER”: PARTE STATICA

Quali funzioni statiche notevoli ospita ?

- **Conversione stringa / numero**
- **Conversione numero / stringa**

In effetti, una variabile di un tipo primitivo *non è un oggetto*, quindi non è possibile invocare su di essa un metodo nello stile "a invio di messaggi" !

Occorre tornare allo stile "a funzioni" classico:

- **NON** `valoreprimitivo.convertitiIn()`
- **MA** `converti(valoreprimitivo, stringa)`

CLASSI “WRAPPER”: PARTE STATICA

Conversione stringa / numero

- **Funzioni della forma `parseXXX`**
  Ad esempio, per gli interi,

```
int x = Integer.parseInt("-1234");
```

  Questa *funzione statica* della classe ***`Integer`*** *analizza una stringa* fornita come parametro e, *se rappresenta un valore intero*, **produce un `int`** inizializzato a tale valore e lo restituisce (come la `atoi` del C).

CLASSI "WRAPPER": PARTE STATICA

E se non è un numero?

- Ad esempio, per gli interi,

```
int x = Integer.parseInt("-ciao");
```

o anche

```
int x = Integer.parseInt("7.54");
```

Cosa succede in questi casi?

Errore: la conversione *non ha senso,* è una contraddizione che *deve essere segnalata.*

CONVERSIONI STRINGA / NUMERO

Funzioni di conversione da stringa ai vari tipi di numeri:

Tipo del risultato	Funzione di conversione
byte	Byte.parseByt

Versioni con specifica della *base di conversione* (per tipi interi):

Tipo del risultato	Funzione di conversione
byte	Byte.parseByte(St

CLASSI "WRAPPER": PARTE STATICA

Oltre alle funzioni *`parseXXX`*, che restituiscono un valore primitivo, esistono anche le funzioni (statiche) `valueOf`, che restituiscono *un oggetto della classe wrapper*

- Ad esempio, per gli interi,

```
Integer wx = Integer.valueOf("-1234");
```

Il risultato non è più un valore **`int`** (come nel caso di **`parseInt`**) ma bensì *una istanza di* ***`Integer`***

CONVERSIONI STRINGA / NUMERO

Versioni che restituiscono *un oggetto della classe wrapper*:

Tipo del risultato	Funzione di conversione
Byte	`Byte.valueO`

La funzione `valueOf` esiste anche per i booleani: restituisce un oggetto `Boolean` che incapsula il valore `boolean` corrispondente alla stringa passata come parametro.

CLASSI "WRAPPER": PARTE STATICA

Conversione numero / stringa

- **Per gli oggetti**, la fa il **metodo `toString`**
  E per i tipi primitivi?
- Vi siete mai chiesti come fa **`println`** a stampare `int`, `float`...?
- E come fa il **+** a concatenare `int`, `float`...a stringhe??

Esiste una *funzione statica* che svolge la stessa operazione per i tipi primitivi... *e si chiama anche lei `toString`!!!*

TIPI PRIMITIVI E CLASSI "WRAPPER"

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

- **<u>versione metodo</u>:**

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

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

- **<u>versione funzione statica</u>:** *(non esiste per Boolean)*

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

  - prende un valore **`int`** e ne produce la rappresentazione sotto forma di stringa
  - è implementata tipicamente appoggiandosi all'altra

ESEMPIO

```
public class EsempioWrapper3 {
 public static void main(String args[]){
   int
   In
   x

   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`**)

UN ESERCIZIO… correlato

ESERCIZIO

Problema

Scrivere una funzione che scambi due interi

- non opera su oggetti → funzione statica
- scritta dentro a una classe contenitore

Quale signature?

```
public static void scambia(... , ... )
```

Forse questa?

```
public static void scambia(int, int)
```

NO: il passaggio sarebbe per valore!

ESERCIZIO

Serve un *passaggio per riferimento.*

Come ottenerlo in Java su un tipo primitivo??

- Java non offre scelta: i tipi primitivi passano *sempre e solo per valore!*
- Solo gli oggetti passano per riferimento.

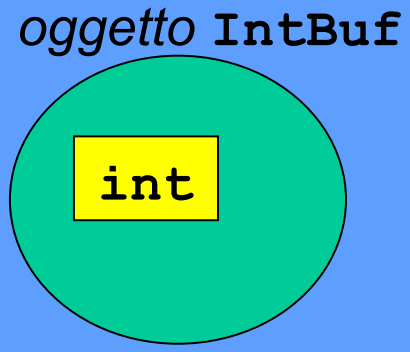
Ma allora...

…basta definire una classe che *incapsuli il tipo primitivo*, e poi passare a scambia *oggetti di quel tipo:*

```
public static void scambia(IntBuf, IntBuf)
```

LA CLASSE `IntBuf`

Definiamo una classe `IntBuf` che
incapsuli il tipo primitivo `int`

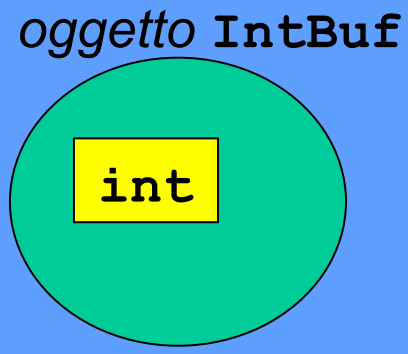


- **Costruttore:** costruisce un oggetto `IntBuf` a partire da un `int`
- **Metodo `getValue`:** recupera il valore `int` da un oggetto `IntBuf`
- **Metodo `setValue`:** cambia il valore `int` contenuto in un oggetto `IntBuf`

```
public class IntBuf {
 private int val;
 public IntBuf(int v) { val = v; }
 public int  getValue() { return val;}
 public void setValue(int v) { val = v;}
}
```

LA FUNZIONE `scambia`

La funzione `scambia` può quindi lavorare su due oggetti `IntBuf`

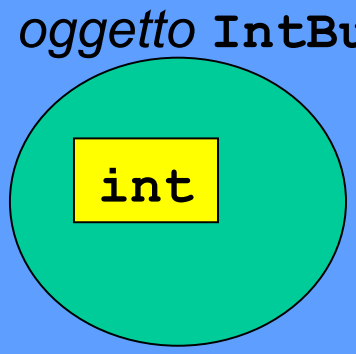


Per scambiare i valori di due oggetti `a` e `b`

- si salva in un temporaneo il valore dell'oggetto `a`
- si cambia il valore dell'oggetto `a` rendendolo pari a `b`
- si cambia il valore dell'oggetto `b` rendendolo pari all'intero temporaneo che ospita il vecchio valore di `a`

LA FUNZIONE `scambia`

La funzione `scambia` può quindi lavorare su due oggetti `IntBuf`




```
public class MyLib {
 public static void scambia(IntBuf a, IntBuf b){
  int temp = a.getValue();
  a.setValue(b.getValue());
  b.setValue(temp);
 }
}
```

UN MAIN DI PROVA

```
public class Prova {
 public static void main(String args[]){
  int x = 10, y = 30;
  System.out.println("x, y = " + x + ", " + y);
  IntBuf a = new IntBuf(x), b = new IntBuf(y);
  System.out.println("a, b = " +
                a.getValue() + ", " + b.getValue());
  MyLib.scambia(a,b);
  System.out.println("a, b = " +
                a.getValue() + ", " + b.getValue());
 }
}
```

UN MAIN DI PROVA

```
public class Prova {
 public static void main(String args[]){
   int x = 10, y = 30;
   System.out.println("x, y = " + x + ", " + y);
   IntBuf a = new IntBuf(x), b = new IntBuf(y);
   System.out.println("a, b = " + a.getValue() + ", "+ b.getValue());
   MyLib.scambia(a,b);
   System.out.println("a, b = " + a.getValue() + ", "+ b.getValue());
 }
}
```

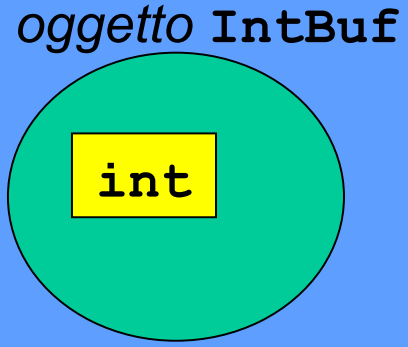
```
C:\temp>java Prova

x, y = 10, 30
a, b = 10, 30
a, b = 30, 10
```

UNA RIFLESSIONE

La classe `IntBuf` prevede un metodo che *modifica il valore:*

- **Costruttore:** costruisce un oggetto `IntBuf` a partire da un `int`
- **Metodo `getValue`:** recupera il valore `int` da un oggetto `IntBuf`
- **Metodo `setValue`:** cambia il valore `int` contenuto in un oggetto `IntBuf`

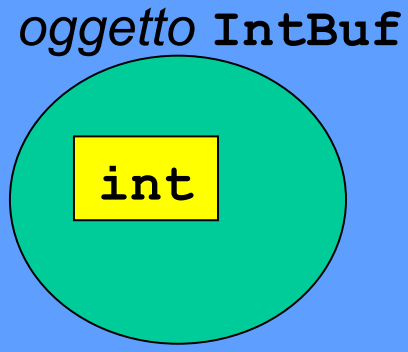



```
public class IntBuf {
 private int val;
 public IntBuf(int v) { val = v; }
 public int  getValue() { return val;}
 public void setValue(int v) { val = v;}
}
```

UNA RIFLESSIONE

Dunque:

- **gli oggetti `IntBuf` non sono *valori*, sono *contenitori***
- **il metodo `setValue` appartiene alla categoria dei *trasformatori***




```
public class IntBuf {
 private int val;
 public IntBuf(int v) { val = v; }
 public int  getValue() { return val;}
 public void setValue(int v) { val = v;}
}
```

CLASSI WRAPPER

Java prevede già *classi standard* per incapsulare valori primitivi: le classi *"wrapper"*

- **esse però sono *valori***
- ***non hanno trasformatori***

Il costruttore costruisce un oggetto "wrapper" a partire da un valore primitivo

Un metodo permette di recuperare il valore primitivo incapsulato in un oggetto "wrapper"

Dunque, le classi *"wrapper"* di Java *non* sono adatte a fungere da contenitori per ottenere il passaggio per riferimento di tipi primitivi.