DIETI UNINA UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE
Scuola Politecnica e delle Scienze di Base
Università degli Studi di Napoli Federico II

Corso di
Ingegneria del Software

I tipi in Java, Wrapper, Enumerazioni, Array e Casting

Sommario

- Tipi fondamentali;
- Array;
- Tipi Wrapper ;
- Boxing, Autoboxing e Unboxing;
- Enumerazione;
- Casting.

::.. Tipi Primitivi (1/4)

Java è un linguaggio tipizzato, ciò significa che ogni variabile prima di essere utilizzata deve essere dichiarata andando ad assegnarle un nome ed un tipo. Il tipo di una variabile può essere costruito dallo sviluppatore per composizione a partire da un set predefinito di tipi detti comunemente tipi primitivi.

Ciascuno di essi è pensato per rappresentare un certo tipo di informazione e utilizzando una quantità specifica di memoria. Inoltre, per ogni tipo è definito un valore di default, che viene impiegato per inizializzare una variabile di quel tipo se il programmatore non ha definito un proprio valore di inizializzazione.

::.. Tipi Primitivi (2/4)

Tipi predefiniti in Java:

Tipo	Q.tà di Memoria	Informazione rappresentata	Valore di Default
byte	8 bit	Numero con segno (con rappresentazione complemento a due) in un range $[-2^{7},(2^{7}-1)]$	0
short	16 bit	Numeri interi (con segno) in un range $[-2^{15},(2^{15}-1)]$	0
int	32 bit	Numeri interi (per default con segno, signed) in un range $[-2^{31},(2^{31}-1)]$	0
long	64 bit	Numeri interi (per default con segno, signed) in un range $[-2^{63},(2^{63}-1)]$	0L
float	32 bit	Numeri in virgola mobile in singola precisione rappresentati con segno, mantissa esponente.	0.0f
double	64 bit	Numeri in virgola mobile in doppia precisione.	0.0d
boolean	N/S [1 bit]	Valori booleani	false
char	16 bit	Caratteri del charset Unicode	\u0000

::.. Tipi Primitivi (3/4)

In Java, accanto agli 8 tipi primitivi sono da considerarsi tipi di dato speciali (detti comunemente Simple Data Objects) anche i tipi **String** e **Number** (e derivati) che fungono in qualche modo da controparte dei dati primitivi dove ci sia l'esigenza di utilizzare un **oggetto** invece che direttamente una variabile in un tipo predefinito.

Gli oggetti di tipo String sono sequenze di char che possono essere inizializzate utilizzando le virgolette (doppi apici):

```
String author = "Douglas Noël Adams";
```

mentre i tipi **Integer**, **Byte**, **Long**, **Float** e **Double** (controparti dei medesimi tipi primitivi scritti con la prima lettera minuscola) sono inizializzabili con i medesimi literals presentati per i corrispondenti tipi nativi.

::.. Tipi Primitivi (4/4)

La differenza consiste nel fatto che i tipi primitivi sono impiegati per la definizione di variabili, inizializzabili implicitamente con il valore di default del tipo di appartenenza.

Gli oggetti sono rappresentati in memoria come una coppia reference e oggetto, con il programmatore che gestisce direttamente i reference ed indirettamente gli oggetti per mezzo di un reference. La dichiarazione:

```
String s;
```

definisce un reference a un tipo String, che viene implicitamente inizializzato a null, indipendentemente dal tipo di appartenenza.

::.. Array (1/6)

Un array in Java è un contenitore che permette di gestire una sequenza di lunghezza fissa di elementi tutti del medesimo tipo. La sintassi per la dichiarazione di una variabile di tipo array è la seguente:

```
Tipo[] nome;        oppure        Tipo nome[];
```

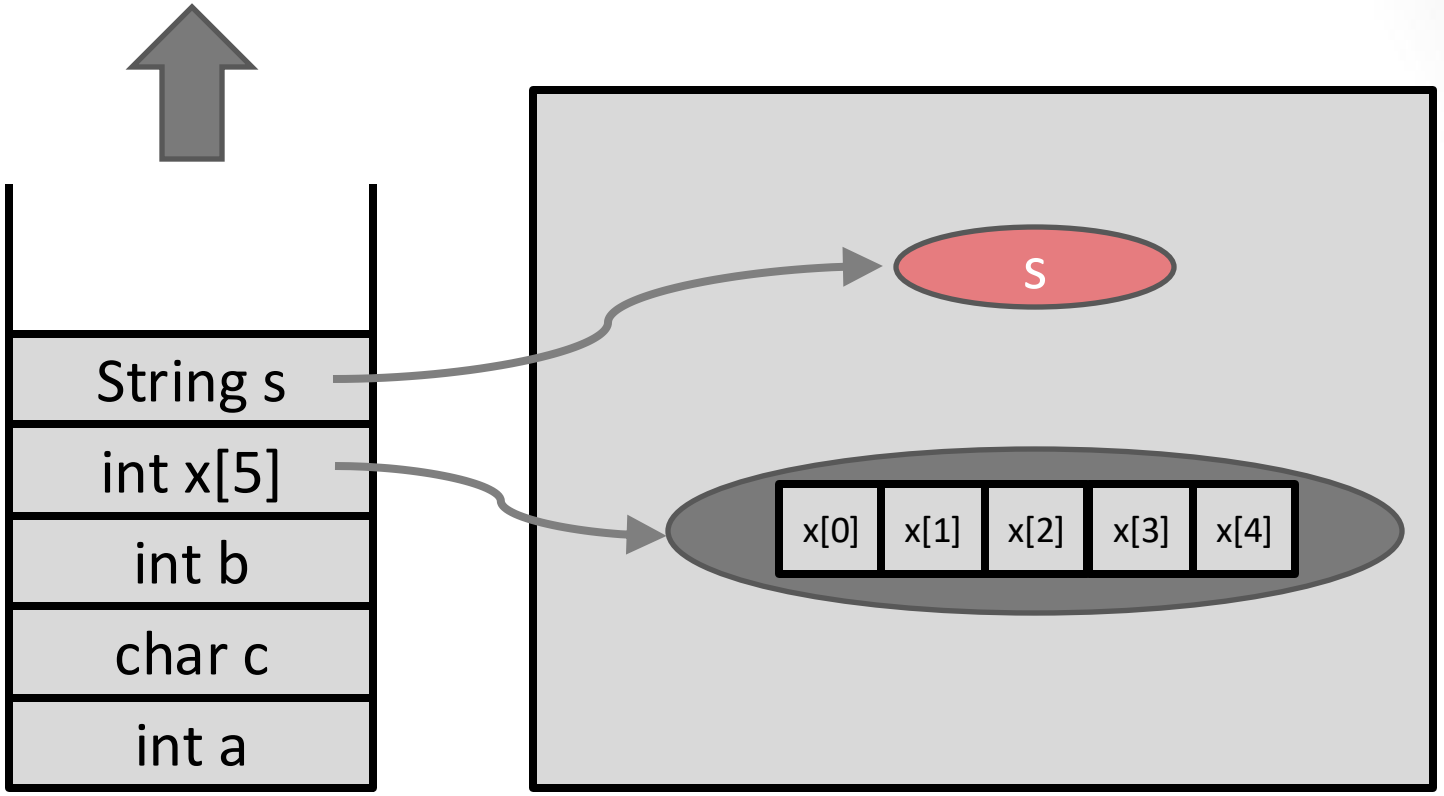
Tipo può essere sia un tipo primitivo, sia una classe. Per default le variabili di tipo array sono inizializzate con il valore null. Quindi, prima di poterle usare, bisogna inizializzarle allocando la memoria per mezzo di new:

```
nome = new Tipo[n];
```

Tutti gli elementi dell'array sono inizializzati con il valore di default previsto dal tipo.

Il numero di elementi in un array, detto lunghezza dell'array, deve essere dichiarato al momento della sua allocazione e non può essere cambiato.

::.. Array (1/6)



STACK
(tipi primitivi, riferimenti)

HEAP
(oggetti)

::.. Array (2/6)

Una volta creato l'array, possiamo accedere ai singoli elementi indicandone la posizione (detta indice) grazie all'operatore '[]':

```
nome[3];
```

L'indice parte da zero fino a lunghezza-1. Si può accedere alla lunghezza dell'array utilizzando la proprietà length che è presente in ogni array.

```
int[] numeroGiorniPerMese = new int[12];

giorniPerMese[0] = 31;

giorniPerMese[1] = 28; // etc ...

giorniPerMese[12] = 31;
```

L'inizializzazione dell'array è decisamente scomoda, ma c'è una sintassi più diretta:

```
int [] numeroGiorniPerMese = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

::.. Array (2/6)

Una volta creato l'array, possiamo accedere ai singoli elementi indicandone la posizione (detta indice) grazie all'operatore '[]':

```
nome[3];
```

L'indice parte da zero fino a lunghezza-1. Si può accedere alla lunghezza dell'array utilizzando la proprietà length che è presente in ogni array.

```
int[] numeroGiorniPerMese = new int[12];

giorniPerMese[0] = 31;

giorniPerMese[1] = 28; // eta

giorniPerMese[
```

Tutti i valori di un array devono essere del tipo specificato per l'array (o ad esso assegnabile).

L'inizializzazione d[illegible]y è decisamente scomoda, ma c'è una sintassi p[illegible]iretta:

```
int [] numeroGiorniPerMese = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

::.. Array (3/6)

Quando utilizziamo gli array è nostra responsabilità non tentare di accedere ad elementi esterni al range definito. Ad esempio:

```
int l = 5;

int [] a = new int[l];

a[9] = 10;
```

Quando mandiamo in esecuzione questo pezzo di codice, esso genera un errore a runtime (non di compilazione): la JVM, quando proviamo ad accedere al decimo elemento dell'array, solleva un'eccezione di tipo ArrayIndexOutOfBoundsException.

Java permette anche l'utilizzo di array di array (detti anche array multi-dimensionali) di profondità arbitraria (o con numero di dimensioni arbitrario) e la sintassi per la loro dichiarazione ed allocazione si ottiene semplicemente ripetendo le parentesi quadre tante volte quante il numero di dimensioni.

::.. Array (4/6)

```
int [][][] arrayConDimensione3 = new int[4][5][6];
```

Analogamente a quanto detto per gli array unidimensionali, Java prevede una sintassi speciale per l'inizializzazione degli array multi-dimensionali:

```
float[][] mat = new int[][] = {
            { 1, 2, 3 },
            { 4, 5, 6 },
            { 7, 8, 9, 10, 11 }
          };
```

L'ultima riga mostra anche come sia possibile creare array multidimensionali '**ragged**', cioè nei quali i sotto-array non abbiano tutti la medesima lunghezza.

::.. Array (5/6)

Gli array sono un costrutto classico di praticamente ogni linguaggio di programmazione e, nonostante il limite notevole di non poter cambiare dimensione (size) dopo la creazione (e qualche complicatezza sintattica degli array multidimensionali), il loro utilizzo è estremamente comune in molti ambiti.

Perciò Java mette a disposizione la classe java.lang.Arrays con numerosi algoritmi per operare sugli array:

- **ricerca;**
- **ordinamento;**
- **copia.**

e altri strumenti per la manipolazione, tutti sotto forma di metodi statici.

::.. Array (6/6)

Esempio di ricerca di un elemento in un array:

```
Arrays.asList(yourArray).contains(yourValue);
```

oppure

```
Arrays.binarySearch(yourArray, elem);
```

Esempio di ordinamento di un array:

```
Arrays.sort(yourArray);
```

Esempio di copia di un array:

```
T[] copy = Arrays.copyOf(yourArray,
        yourArray.length);
```

dove T è il tipo degli elementi di yourArray.

::.. Tipi Wrapper (1/4)

In Java per ogni tipo primitivo esiste un corrispondente Simple Data Object o, come si suol dire, una Classe Wrapper. Dato un tipo primitivo (i cui nomi iniziano tutti rigorosamente con la prima lettera minuscola) si ottiene il corrispondente Data Object sostanzialmente capitalizzando il nome come mostrato nella tabella seguente:

Tipo Primitivo	Tipo Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

::.. Tipi Wrapper (2/4)

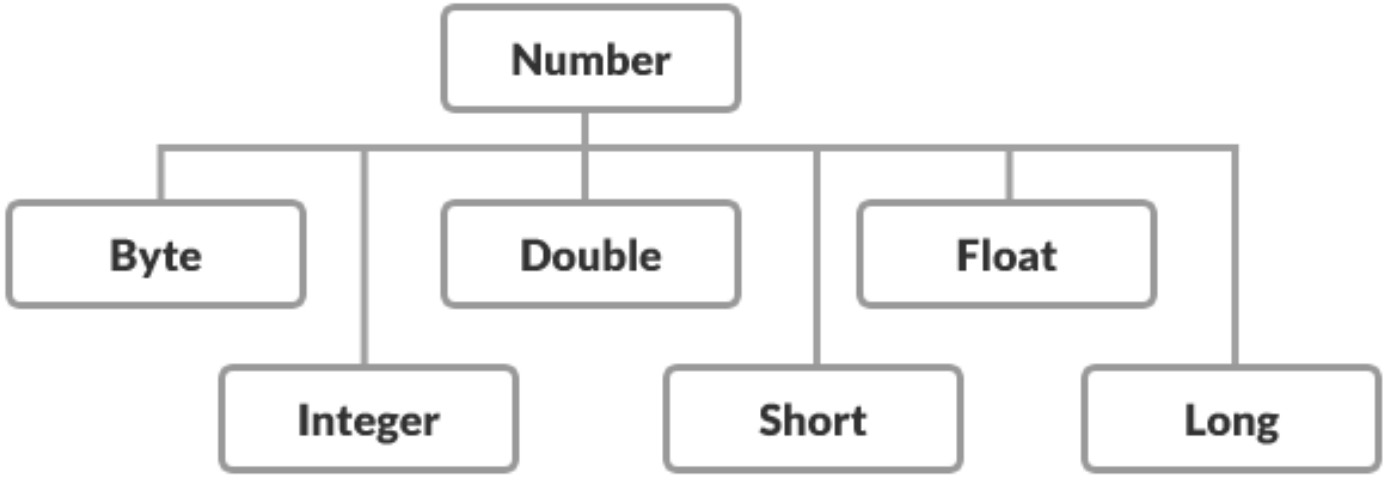
Ad un primo sguardo può sembrare che ci sia poca differenza tra un tipo primitivo e la sua controparte 'wrapped' (spesso detta 'boxed'); tra le due c'è in realtà una fondamentale distinzione: i tipi primitivi non sono usati per definire oggetti e non hanno associata alcuna classe e quindi devono essere trattati in modo diverso rispetto agli altri tipi 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.

::.. Tipi Wrapper (3/4)

Tutte le classi wrapper sono definite nel package java.lang e sono qualificate come final, perciò non è possibile derivare da loro. Inoltre tutte queste classi sono immutabili, cioè non è possibile dopo la costruzione cambiarne il valore.

Mentre 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.



::.. Tipi Wrapper (4/4)

Creazione di oggetti di tipo wrapper a partire da variabili di tipo predefinito:

```
int val = 44;

// dato una variabile di un tipo primitivo
// si crea l'istanza del relativo wrapper
Integer value = new Integer(val);

// dall'oggetto è possibile "estrarre" il valore
int valueBack = value.intValue();
```

::.. Parsing

Le classi wrapper dispongono di utilissimi metodi per fare il parsing di stringe in valori numerici:

```
String quarantatre = "43";
Integer q = new Integer(quarantatre);
```

Quando utilizziamo questi metodi per la conversione occorre gestire una eventuale situazione di errore:

```
String quarantaquattro = "quarantaquattro";
Integer q = new Integer(quarantaquattro);
```

Il parser non sarebbe in grado di processare con successo la stringa e otterremmo un errore che la JVM segnala attraverso una eccezione di tipo **NumberFormatException**.

::.. Boxing, unboxing e autoboxing (1/6)

Originariamente, per convertire una variabile in un tipo primitivo nella sua corrispondente classe wrapper, Java richiedeva l'invocazione esplicita del costruttore o di funzioni di parsing:

```
Integer x = new Integer (10);

Double y = new Double (5.5);

Boolean z = Boolean.parseBoolean("true");
```

Queste operazioni sono note come operazioni di boxing, cioè "inscatolamento" del tipo primitivo nel relativo tipo wrapper al fine di utilizzare un oggetto e tutte le sue proprietà.

::.. Boxing, unboxing e autoboxing (2/6)

Lo **Autoboxing** è una caratteristica del linguaggio che, a partire dalla versione 1.5, ci consente di lavorare con tipi primitivi e tipi wrapper in maniera **intercambiabile**. Vediamo questa "novità" con un esempio:

```
Integer x = 10;

Double y = 5.5f;

Boolean z = true;

Number n = 0.0f;
```

Attraverso autoboxing gli oggetti vengono **automaticamente creati** con i valori di riferimento passati in fase di inizializzazione, senza generare errori. Questo permette di scrivere codice più leggibile e maneggevole.

::.. Boxing, unboxing e autoboxing (3/6)

Chiaramente alla funzione di boxing è associata l'operazione duale di unboxing che trae gli stessi vantaggi della precedente:

```
int x = -1;
Integer y = x;
int y = y.intValue();
```

Il linguaggio si arricchisce, permettendo allo sviluppatore di non preoccuparsi delle operazioni di conversione (boxing e unboxing, appunto) lasciandole al compilatore del bytecode che si occuperà di gestirle per noi (autoboxing).

```
Double d = new Double(5.5);
double pi = d;
```

::.. Boxing, unboxing e autoboxing (4/6)

Dal punto di vista della sintassi, l'autoboxing e unboxing produce notevoli vantaggi in quanto è possibile associare gli operatori aritmetici e le strutture condizionali ai tipi wrapper:

```
Integer x = 0;
Boolean verify = true;
while (verify){
  x++;
  if (x > 10)
    verify = false;
}
```

Unica eccezione è il **test di uguaglianza** tra due istanze di wrapper contenenti il medesimo valore.

::.. Boxing, unboxing e autoboxing (5/6)

```
Integer x = 1000;
Integer y = 1000;


if (x==y) {
  // ??
}
```

La condizione risulta falsa, perché si fa la comparazione tra due istanze di oggetto senza effettuare unboxing.

::.. Boxing, unboxing e autoboxing (6/6)

Altro comportamento particolare è l'overload di metodi, dove c'è la possibilità di confusione:

```
public void metodoA(Integer x);

public void metodoA(double y);

public void usaA(){

    int d = 0;

    metodoA(d);

}
```

In questo caso il compilatore non effettua alcuna operazione di unboxing, e la chiamata sarebbe fatta al metodo che accetta come tipo il double.

In generale, di fronte a situazioni di ambiguità, per mantenere anche la compatibilità con le precedenti versioni, il comportamento è quello che ci sarebbe con la versione di Java 1.4.

::.. Casting

Java è un linguaggio "fortemente tipizzato", perciò i tipi sono fondamentali e c'è uno stretto controllo sul loro utilizzo. Ad esempio, non è possibile assegnare un carattere ad un intero, cosa possibile nel linguaggio C grazie alla conversione implicita, ma è anche impossibile assegnare un double ad un float, senza un esplicito casting. Il casting implicito è permesso solo da tipi più «piccoli» (ad es. byte) a tipi più grandi)ad es. int.

Il casting consiste nel forzare un valore di un tipo ad assumere un tipo diverso: se scriviamo 5 ad esempio abbiamo un numero intero, ma se ne facciamo un cast a float intenderemo la versione di 5 "reale", per farlo basta scrivere:

```
(float) 5;
```

Pertanto le assegnazioni di valori e variabili a variabili che non concordano nel tipo è possibile solo esplicitando l'operazione di casting.

::.. Enumerazione (1/9)

A partire dalla versione 5, in Java è stato introdotto uno speciale tipo chiamato Enum o Enumerated Type che, vincola una variabile ad assumere solo un determinato insieme di valori.

Immaginiamo di scrivere un programma che gestisca un calendario. Ci servirà un modo per identificare i giorni della settimana. Possiamo definire la variabile giornoDellaSettimana come int e stabilire che 0 corrisponde a Lunedì, 1 a Martedì e così via:

```
int giornoDellaSettimana = 4; // VEN per convenzione
```

e con quache costante (static final) potremmo anche rendere leggibile il codice:

```
public static final int LUN = 0;
// ...
public static final int VEN = 4;
int giornoDellaSettimana = VEN;
```

::.. Enumerazione (2/9)

Purtroppo, per il compilatore, giornoDellaSettimana è una variabile int e quindi non potrà mai verificare che non ci sia mai nel nostro codice una linea in cui viene assegnato il valore 9 (o negativo). Per rappresentare un giorno della settimana, definiamo un tipo ad hoc:

```
public enum Giorno {
    LUNEDI,
    MARTEDI,
    MERCOLEDI,
    GIOVEDI,
    VENERDI,
    SABATO,
    DOMENICA // opzionalmente può terminare con ";"
}
```

::.. Enumerazione (3/9)

```
Giorno giornoDellaSettimana;
```

Così avremo una variabile che potrà contenere solamente un valore appartenente al set specificato nella definizione dell'enum Giorno e contemporaneamente avremo anche i nomi simbolici (le costanti di prima) da usare nella scrittura del programma:

```
giornoDellaSettimana = Giorno.VENERDI;
```

Tecnicamente, una enum è una classe come le altre ma che implicitamente estende sempre la classe java.lang.Enum. Ciò rendere impossibile avere enum che derivino da altri tipi. Il vantaggio è che enum dispone implicitamente del metodo statico values() che ritorna un array di tutti i possibili valori che potranno assumere le variabili cha avranno come tipo l'enum.

::.. Enumerazione (4/9)

Analogamente il compilatore ci offre la possibilità di convertire stringhe in valori del nostro enum:

```
Giorno g = Giorno.valueOf("SABATO");
```

Tale istruzione assegnerà alla variabile g il valore Giorno.SABATO. Attenzione: la stringa deve avere il medesimo case e non può contenere spazi, infatti l'esecuzione (non la compilazione) di:

```
Giorno g = Giorno.valueOf("Sabato");
```

genererà un'eccezione di tipo:

"java.lang.IllegalArgumentException".

In generale infine, il fatto che le enum siano a tutti gli effetti classi, apre la possibilità di aggiungere dentro di essi metodi e field e, e questo è più sorprendente, anche costruttori.

::.. Enumerazione (5/9)

Se la definizione dell'enum contiene metodi e/o variabili membro:

- **La lista degli enumeratori deve essere terminata da `';'` (che altrimenti non è obbligatorio);**
- **Gli eventuali costruttori devono essere privati (o package private) e non possono essere chiamati esplicitamente, sono unicamente a disposizione del compilatore;**
- **La lista degli enumeratori, nel caso in cui siano definiti dei costruttori, non deve essere considerata come una lista di etichette ma come una forma compatta per istruire il compilatore a costruire determinate istanze della classe ed assegnare loro un nome simbolico;**
- **Non ci sono restrizioni circa i metodi e variabili che possono essere inclusi nel corpo di un enum.**

::.. Enumerazione (6/9)

Immaginiamo di dover gestire in un programma gli elementi chimici, potremmo probabilmente costruire un enum che li contenga tutti. Ad ogni elemento, però, dobbiamo associare alcune informazioni aggiuntive, diciamo numero atomico e massa atomica, e predisporre opportuni metodi per accedervi:

```
private int numeroAtomico;

private double massaAtomica;

private String simbolo;

public int getNumeroAtomico() {

    return numeroAtomico;

}

// ... altri getter/setter
```

e un costruttore per permettere l'inizializzazione delle variabili membro.

::.. Enumerazione (7/9)

```
private Elemento(String simbolo, int numeroAtomico,
double massaAtomica) {
    this.simbolo = simbolo;
    this.numeroAtomico = numeroAtomico;
    this.massaAtomica = massaAtomica;
}
```

Se a questo punto tentassimo di definire i valori dell'enum come abbiamo fatto per i giorni:

```
public enum Elemento {
    IDROGENO,
    ELIO,
    // ... ;}
```

Otterremmo un errore di compilazione che ci avviserebbe che il costruttore default non esiste.

::.. Enumerazione (8/9)

Il costruttore definito precedentemente prevede che vangano specificati alcuni parametri, così siamo obbligati a costruire i nostri valori nell'enum, con il costruttore che abbiamo fornito:

```
public enum Elemento {
  IDROGENO("H", 1, 1.008),
  ELIO("He", 2, 4.003),
  // ... altri elementi ...
  LITIO("Li", 3, 6.491);
  // variabili e funzioni membro, e costruttore
  private Elemento(String simb, int num, double m)
  { ... }
}
```

```
Elemento var = Elemento.IDROGENO;

String peso = var.getMassaAtomica();
```

::.. Enumerazione (8/9)

Il costruttore definito precedentemente prevede che vangano specificati alcuni parametri, così siamo obbligati a costruire i nostri valori nell'enum, con il costruttore che abbiamo fornito:

```
public enum Elemento {
   IDROGENO("H", 1, 1.008),
   ELIO("He", 2, 4.003),
   // ... altri elementi ...
   LITIO("Li", 3, 6.491);
   // variabili e funzioni
   private Elemento(String
   { ... }
}
```

Le variabili di tipo Elemento, per costruzione, saranno sempre elementi validi e con le proprietà che ci interessano sempre inizializzate e disponibili.

```
Elemento var = Elemento.IDROGENO;

String peso = var.getMassaAtomica();
```

::.. Enumerazione (9/9)

Enum risolve quindi un'esigenza reale, quella di definire un insieme di valori predefiniti, senza ricorrere alla mediazione di costanti intere, con la possibilità di avere una classe. Le enumerazioni possono essere utilizzate nei cicli for-each e nei costrutti switch. Il metodo toString(), di default, è uguale al nome assegnato alla variabile.

I tipi definiti in una enumerazione sono istanze di classe, non tipi interi. I valori di una enumerazione sono public final static, quindi immutabili. Il metodo == è sovrascritto, quindi può essere usato in maniera intercambiabile al metodo equals. Esiste la coppia di metodi valueOf()/toString() che possono essere sovrascritti.