

Tipi primitivi e variabili

esempi

◆ Tipi numerici:

- **byte**: 8 bit
- **short**: 16 bit
- **int**: 32 bit
- **long**: 64 bit
- **float**: 32 bit
- **double**: 64 bit

```
byte un_byte;
int a, b=3, c;
char c='h', car;
boolean trovato=false;
```

A differenza di C/C++, per
ciascun tipo è definito un
valore di default

◆ Altri tipi:

- **boolean**: `true` o `false`
- **char**: 16 bit, carattere Unicode

I tipi “riferimento”

- ◆ Tipi **array**
- ◆ Tipi definiti dall'utente (o predefiniti)

- **classi**
- **interfacce**

Java non supporta **struct**
(né tantomeno **union**)

tipo
(classe)

nome variabile
(oggetto)

operatore
di creazione

costruttore
(associato alla classe)

`Point punto = new Point(10,10);`

Tipi array

- ◆ Sono anch'essi ***tipi riferimento***, come le classi
- ◆ Dato un tipo **T** (predefinito o utente) un array di **T** è definito come **T []**
- ◆ È possibile dichiarare array multidimensionali:

T [] [] T [] [] []

...

```
int[] ai1, ai2;  
float[] af1;  
double ad[];  
Persona[][] ap;
```

```
int[] ai = {1,2,3};  
double[][] ad = {{1.2, 2.5}, {1.0, 1.5}}
```

Tipi array: allocazione di memoria

- ◆ In mancanza di inizializzazione, la dichiarazione di un array **non** alloca spazio per i suoi elementi
- ◆ L'allocazione si realizza **dinamicamente** tramite l'operatore:

new <tipo> [<dimensione>]

```
int[] i = new int[10], j = {10,11,12};  
float[][] f = new float[10][10];  
Persona[] p = new Persona[30];
```

- ◆ Se gli elementi non sono di un tipo primitivo, l'operatore **new** alloca solo lo spazio per i riferimenti

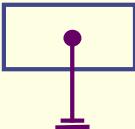
Array di oggetti: definizione

A

```
Person[ ] person;  
person = new Person[20];  
person[0] = new Person( );
```

È definita solo la variabile **person**. L'array vero e proprio non esiste

person



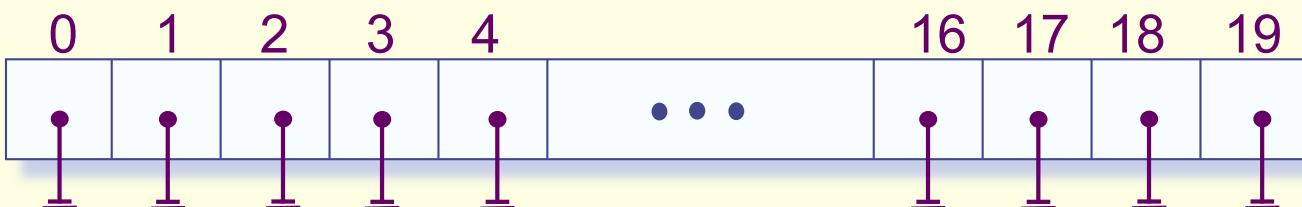
Array di oggetti: definizione

B

```
Person[ ] person;  
person = new Person[20];  
person[0] = new Person( );
```

Ora l'array è stato
creato ma i diversi
oggetti di tipo **Person**
non esistono ancora

person



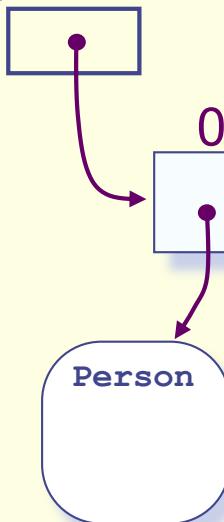
Array di oggetti: definizione

```
Person[ ] person;  
person = new Person[20];  
person[0] = new Person( );
```

C

Un oggetto di tipo
Person è stato creato
e un riferimento a tale
oggetto è stato inserito
in posizione 0

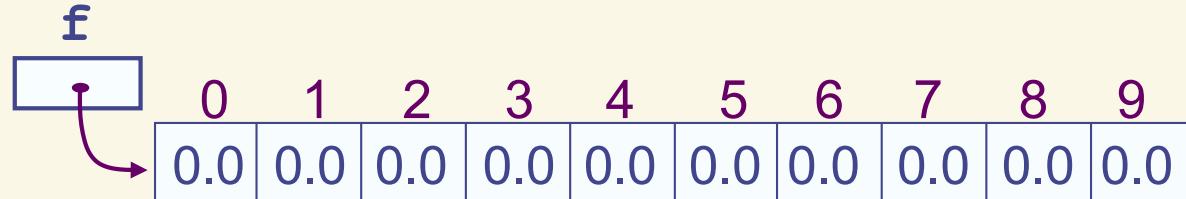
person



Array di oggetti vs. array di tipi base

L'istruzione:

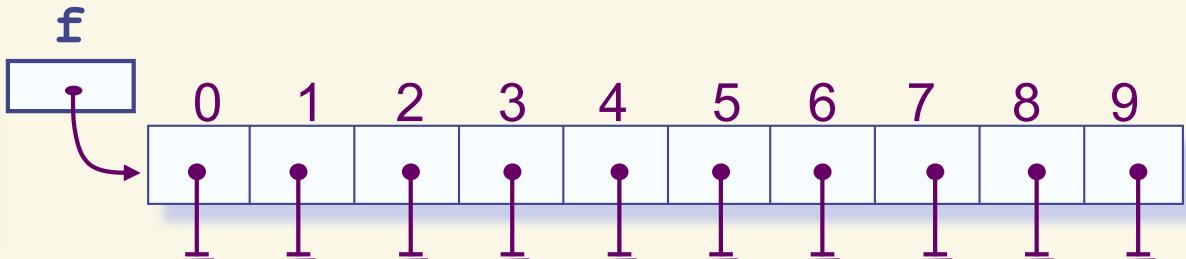
`float f[] = new float[10];`
crea un oggetto di tipo array di `float` e alloca spazio per 10 `float`



L'istruzione:

`Person p[] = new Person[10];`
crea un oggetto di tipo array di `Person` e alloca spazio per 10 riferimenti a oggetti di tipo `Person`

Non viene allocato
spazio per gli oggetti
veri e propri



La Pila in Java

Ogni classe appartiene a un
package (ci torneremo più avanti)

```
package strutture;
```

```
public class Pila {  
    int size;  
    int defaultGrowthSize;  
    int marker;  
    int contenuto[];
```

```
Pila(int initialSize) {  
    size = initialSize;  
    defaultGrowthSize = initialSize;  
    marker = 0;  
    contenuto = new int[initialSize];  
}  
...
```

in C++

```
class Pila {  
private:  
    int size;  
    int defaultGrowthSize;  
    int marker;  
    int *contenuto;
```

in C++

```
Pila::Pila(int initialSize) {  
    size = initialSize;  
    defaultGrowthSize = initialSize;  
    marker = 0;  
    contenuto = new int[initialSize];  
}
```

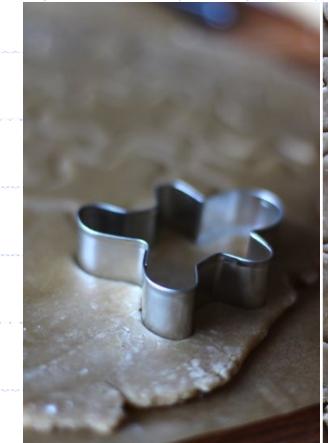
Classi e oggetti: attributi e metodi

- ◆ Gli **attributi** costituiscono lo **stato** degli oggetti istanziati a partire dalla classe
 - il loro valore è diverso per ogni oggetto
- ◆ I **metodi** definiscono il **comportamento** degli oggetti istanziati a partire dalla classe
 - il loro codice è lo stesso per ogni oggetto

classe



new



memoria



**oggetti, ognuno
con il suo stato**



Classi e oggetti: riferimenti (*reference*)

- ◆ Le **istanze** di una classe si chiamano **oggetti**
- ◆ Ogni variabile il cui tipo sia una classe (o un'interfaccia) contiene un **riferimento** ad un oggetto
- ◆ In Java, gli oggetti sono accessibili **solo** per riferimento
 - In C++ possono essere dichiarati sullo stack oppure sullo heap, con regole semantiche diverse
- ◆ Ad ogni variabile di tipo riferimento può essere assegnato il riferimento **null**: **Punto p = null;**
- ◆ «*Java non ha i puntatori*»
 - In realtà, ciò che manca in Java è l'**aritmetica dei puntatori** perché un riferimento non contiene un indirizzo: per il resto, un puntatore C/C++ è del tutto analogo a un riferimento Java

Classi e oggetti: regole passaggio parametri

- ◆ I parametri il cui tipo sia uno dei **tipi primitivi** sono passati **per copia**
- ◆ I parametri il cui tipo sia un **tipo riferimento** (classi, interfacce e array) sono passati **per riferimento**
 - ovvero per copia del riferimento
- ◆ Quindi, gli oggetti sono **sempre** passati per riferimento

Creazione e distruzione degli oggetti

- ◆ Nuovi oggetti sono costruiti usando l'operatore **new**
- ◆ La creazione di un oggetto include l'invocazione di un metodo particolare dell'oggetto: il **costruttore**
- ◆ Esso svolge due operazioni fondamentali:
 - **allocazione** della memoria necessaria a contenere l'oggetto
 - **inizializzazione** dello spazio allocato
- ◆ A differenza del C++, in Java non vi è un distruttore, né si devono (o possono!) deallocare esplicitamente gli oggetti...
- ◆ ... perché di ciò si occupa il **garbage collector**

Garbage collection

- Il ***garbage collector*** (GC) interviene *automaticamente* quando serve memoria
 - elimina gli oggetti per cui non vi sono più riferimenti attivi
- Può essere attivato esplicitamente: `System.gc()` ;
 - di norma ***non*** necessario
- È possibile definire nel metodo `finalize()` azioni da eseguire all'atto della distruzione di un oggetto
 - Non è un distruttore!
 - È pensato per poter rilasciare risorse diverse dalla memoria in caso di distruzione dell'oggetto

Classi e costruttori

- ◆ Nella definizione di una classe è possibile specificare uno o più costruttori (*overloading*)
- ◆ Un costruttore ha lo stesso nome della classe, e non indica il tipo del risultato
- ◆ Se non viene specificato nessun costruttore, il compilatore inserisce un **costruttore di default** (senza parametri) che:
 - alloca lo spazio per gli attributi di tipo primitivo e li inizializza al valore di default
 - alloca lo spazio per i riferimenti agli attributi di tipo definito dall'utente, e li inizializza a **null**

Ancora sui costruttori

- ◆ È possibile invocare un costruttore dall'interno di un altro con la notazione:
this(<elenco parametri>);

```
class Persona {  
    String nome;  
    int eta;  
    Persona(String nome) {  
        this.nome = nome;  
        eta = 0;  
    }  
    Persona(String nome, int eta) {  
        this(nome);  
        this.eta = eta;  
    }  
}
```

La Pila in Java

```
private void cresci() {
    size += defaultGrowthSize;
    int temp[] = new int[size];
    for (int k=0; k<marker; k++)
        temp[k] = contenuto[k];
    this.contenuto = temp;
}
```

this contiene un riferimento all'oggetto corrente: il suo uso non è necessario, ma può essere utile per aggirare mascheramenti

Il metodo è **private** e quindi non accessibile da altre classi: ne evita un uso improprio

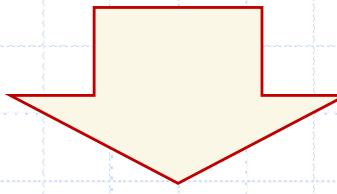
Al contrario del C++, la clausola **private**
1) fa parte della firma del metodo e 2) non è implicita

```
void cresci() {
    size += defaultGrowthSize;
    int *temp = new int[size];
    for(int k=0; k<s->marker; k++)
        temp[k] = contenuto[k];
    delete [] (contenuto);
    this->contenuto = temp;
}
```

in C++

cresci: implementazione alternativa

```
private void cresci() {  
    size += defaultGrowthSize;  
    int temp[] = new int[size];  
    for (int k=0; k<marker; k++)  
        temp[k] = contenuto[k];  
    contenuto = temp;  
}
```



```
private void cresci() {  
    size += defaultGrowthSize;  
    int temp[] = new int[size];  
    System.arraycopy(contenuto, 0, temp, 0, marker-1);  
    contenuto=temp;  
}
```

Using System.arraycopy()

```
System.arraycopy(  
    Object src, int src_position,  
    Object dst, int dst_position, int length  
) ;
```

- Copia **length** elementi dell'array **src**,
a partire dall'elemento con indice
src_position, nell'array **dst**, a partire
dall'elemento con indice **dst_position**

La Pila in Java

in C++

```
int estrai() {  
    assert(marker>0) : "Estrazione da un pila vuota!";  
    return contenuto[--marker];  
}
```

```
int estrai() {  
    assert(marker>0);  
    return contenuto[--(marker)];  
}
```

Output in caso l'asserzione sia falsa

```
java.lang.AssertionError: Estrazione da un pila vuota!  
at pila.Pila.estrai(Pila.java:22)  
at pila.Pila.main(Pila.java:39)
```

Asserzioni in Java

```
assert(marker>0) : "Estrazione da un pila vuota!" ;
```

equivale a

```
if (marker==0) {  
    System.out.println("Estrazione da una pila vuota!");  
    System.exit(1);  
}
```

- ◆ L'uso delle asserzioni va esplicitamente abilitato
 - es., da command line: **java -ea Pila**
- ◆ Il messaggio è opzionale (ma consigliato)

La Pila in Java

```
public static void main(String args[]) {  
    Pila s = new Pila(5);  
    for(int k=0; k<10; k++)  
        s.inserisci(k);  
    for(int k=0;k<12;k++)  
        System.out.println(s.estrai());  
}
```

in C++

```
int main() {  
    Pila *s = new Pila(5);  
    for(int k=0; k<10; k++)  
        s->inserisci(k);  
    for(int k=0; k<12; k++)  
        cout << s->estrai() << endl;  
}
```

System serve da libreria globale

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

`System.gc();`

`System.runFinalization();`

`System.exit(int status);`

`System.arraycopy(...);`

`long System.currentTimeMillis();`

... ma **System** non è
un oggetto!

Metodi e attributi di classe

- ◆ Sintassi:

static <definizione attributo o metodo>

- ◆ Un attributo di classe è condiviso da tutti gli oggetti della classe

- ◆ Si può accedere ad un attributo di classe senza bisogno di creare un oggetto tramite la notazione:

<classe>.<attributo>

- ◆ Un metodo di classe può essere invocato senza bisogno di creare un oggetto tramite la notazione:

<classe>.<metodo>(<par.attuali>)

Attributi di classe: condivisione

- ◆ Un attributo di classe è condiviso da tutti gli oggetti della classe ...
- ◆ ... quindi modifiche apportate attraverso un'istanza sono visibili anche alle altre

```
class C {  
    static int x = 0;  
    int y = 0;  
}  
...  
C a = new C();  
C b = new C();  
a.x = 5;  a.y = 1;  
b.y = b.x + 2;
```

qui **b.y** vale **7**
(e non **2**)

Metodi di classe: vincoli

- ◆ Dall'istanza posso accedere sia a metodi convenzionali che **static**
- ◆ Dalla classe posso accedere solo a metodi **static**
- ◆ Un metodo **static** può accedere ai soli attributi e metodi **static**
- ◆ Un metodo convenzionale può accedere liberamente a metodi ed attributi **static**

```
class Shape {  
    static Screen screen = new Screen();  
    static void setScreen(Screen s) { screen=s; }  
    void show() { ... }  
}  
...  
Shape.setScreen(new Screen()); // corretto  
Shape.show(); // errato  
Shape s1 = new Shape(), s2 = new Shape();  
Screen s = new Screen();  
s1.setScreen(s); // corretto  
s1.show(); // corretto
```

quanto vale qui
s2.screen?

Attributi costanti

- ◆ È possibile definire attributi costanti con la notazione:

```
final <definizione di attributo>=<valore>
```

- ◆ Vedremo più avanti che il modificatore **final** ha anche altri usi (e implicazioni)
- ◆ **static** e **final** si possono usare insieme, per definire costanti accessibili a partire dalla classe e non dall'istanza

```
class Automobile {  
    int colore;  
    final int BLU=0, GIALLO=1,...;  
    void dipingi(int colore) { this.colore=colore; }  
}  
...  
Automobile a = new Automobile();  
a.BLU = 128;                                // errato  
System.out.println("BLU=" + a.BLU);      // corretto
```

Domanda...

◆ È corretto affermare che tramite attributi e metodi di classe e attributi costanti è possibile definire funzioni, variabili e costanti globali?

- In un certo senso sì, poiché è possibile accedere a questi elementi da un qualsiasi ambito (si pensi a `System`)
- Tuttavia, tali elementi non si trovano in un unico ambiente globale indistinto (come in C); viceversa, il loro accesso è
 - ◆ modularizzato attraverso le classi
 - ◆ controllato attraverso i package

Package e *information hiding*

◆ Package e visibilità di attributi e metodi

- Attributi e metodi di una classe **MyClass** per cui non è dichiarato alcun tipo di visibilità sono visibili solo nelle classi che appartengono allo stesso package di **MyClass**

◆ Package ed importazione di classi

- Un package contiene un insieme di classi **public** ed un insieme di classi **private**: solo le classi **public** si possono importare in altri package

◆ Package e compilation unit

- Ogni compilation unit (file **.java**) contiene classi appartenenti allo stesso package ...
- ... e risiede in un folder con il nome del package
- Una compilation unit contiene una sola classe **public** (per default la prima) ed eventualmente altre **private**

I costrutti del linguaggio "forzano" una corretta organizzazione dei file

Il package `java.lang`

- ◆ Il package `java.lang` contiene classi di uso molto frequente (`String`, `Object`, ecc.)
- ◆ Non è necessario importare le classi appartenenti al package `java.lang` prima di utilizzarle

Class String

java.lang
Class String

java.lang.Object
|
+--java.lang.String

All Implemented Interfaces:

CharSequence, Comparable, Serializable

```
public final class String
extends Object
implements Serializable, Comparable, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because `String` objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

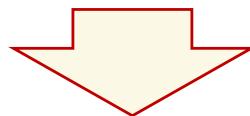
Abbiamo creato una "libreria" ...
ma ne esistono tantissime già
pronte nella distribuzione Java!

Consultate il "javadoc"

Tipi enumerativi

- Come altri linguaggi, anche Java fornisce il costrutto **enum**

```
public static final int APPLE_FUJI      = 0;  
public static final int APPLE_PIPPIN    = 1;  
public static final int APPLE_GRANNY_SMITH = 2;  
public static final int ORANGE_NAVEL   = 0;  
public static final int ORANGE_TEMPLE   = 1;  
public static final int ORANGE_BLOOD    = 2;
```



```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }  
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

- Rende la definizione di costanti più concisa e leggibile
- Raggruppa costanti, con controlli in compilazione, es.
`Apple a = Apple.FUJI; //non posso passare un'arancia...`
- Evita l'uso diretto dei valori delle costanti, es:
`int i = (APPLE_FUJI - ORANGE_TEMPLE)/APPLE_PIPPIN;`

enum in Java

- In Java, **enum** è molto più potente rispetto ad altri linguaggi
- È del tutto analoga a una classe: può avere suoi attributi e metodi, ma...
- ... estende da **java.lang.Enum** ed è **final**: non è possibile estendere una **enum**
 - Può però implementare interfacce
 - Utile da sapere:
 - **toString()** ritorna il nome della costante, utile per le stampe di debugging
 - **values()** ritorna un array contenente le costanti nell'ordine di dichiarazione, utile per realizzare cicli