La Pila in Java

```
package strutture;

public class Pila {
  private int size;
  private int marker;
  private contenuto[];

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

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

in C++

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

in C++

La Pila in Java

```
package strutture;

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

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

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

in C++

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

in C++

Attributi costanti

- È possibile definire attributi costanti con la notazione:

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

- Vedremo più avanti che la il modificatore **`final`** ha anche altri usi (e implicazioni)

```
final int GROWTH_SIZE=5
```

- Per convenzione le costanti in Java sono scritte interamente in lettere maiuscole.

La Pila in Java

this->size

```
void inserisci(int k) {
   const int growthSize=5;
   if(this->marker == this->size)
     this->cresci(growthSize);
   this->contenuto[this->marker] = k;
   this->marker++;
}
```

in C++

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

(quasi) identici!

this.size

```
void inserisci(int k) {
      final int GROWTH_SIZE=5;
      if(this.marker == this.size)
        this.cresci(GROWTH_SIZE);
      this.contenuto[this.marker] = k;
      this.marker++;
   }
```

La Pila in Java

Senza this...

```
void inserisci(int k) {
   const int growthSize=5;
   if(marker == size)
     cresci(growthSize);
   contenuto[marker] = k;
   marker++;
}
```

in C++

proprio identici!

```
void inserisci(int k) {
      final int GROWTH_SIZE=5;
      if(marker == size)
        cresci(GROWTH_SIZE);
      contenuto[marker] = k;
      marker++;
   }
```

La Pila in Java

in C++

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

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

```
private:
void cresci(int delta){;
   size += delta;
   int *temp = new int[size];
   //int temp[] = new int[size];
   for(int k=0; k<marker; k++) {
      temp[k] = contenuto[k];
   }
   delete [](contenuto);
   contenuto = temp;
}
```

in C++

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

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

Non ci va la delete!

La Pila in Java

versione
con il this esplicito

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

in C++

Non ci va la delete!

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

Distruzione degli oggetti

- 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

E finalmente, il main!

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;
}
```

```
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());
  }
}
```

Notiamo nuovamente . invece di ->

Java: Tipi primitivi e variabili

- Tipi numerici:
  - **byte**: 8 bit
  - **short**: 16 bit
  - **int**: 32 bit
  - **long**: 64 bit
  - **float**: 32 bit
  - **double**: 64 bit
- Altri tipi:
  - **boolean**: **true** o **false**
  - **char**: 16 bit, carattere Unicode

esempi

```
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***

```
package strutture;
   …
```

Package e *information hiding*

Ne discuteremo in dettaglio!

- 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
  - Una compilation unit contiene una sola classe **`public`** (per default la prima) ed eventualmente altre **`private`**

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

Creazione 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

Classi e costruttori

Lo discuteremo in dettaglio!

- Nella definizione di una classe è possibile specificare uno o più costruttori (*overloading*)
- Un costruttore ha lo stesso nome della classe, non indica il tipo del risultato e non ha una return
- Viceversa, 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

Lo discuteremo in dettaglio!

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;
  }
}
```

Classi e oggetti: attributi e metodi

- Gli ***attributi*** *(variabili di istanza)* 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, con regole semantiche diverse
- Ad ogni variabile di tipo riferimento può essere assegnato il riferimento **`null`**: `Punto p = null;`

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

Tipi "riferimento"

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

Java non supporta `struct` e `union`

tipo (***classe***)

nome variabile (***oggetto***)

operatore di creazione

costruttore (associato alla classe)

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

Java non ha i puntatori?

Java:

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

C++:

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

Allocato in heap

Allocato in stack;
Non possibile in Java

Quel che non c'è in Java è l'*aritmetica dei puntatori*: per il resto, un puntatore C/C++ è del tutto analogo a un riferimento Java

In C++, dobbiamo distinguere tra:

```
Punto p1;
Punto *p2;
p1.x
p2->x
```

Allocato in stack;
Non possibile in Java

Allocato in heap

In Java, scriviamo

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

intendendo

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

E allora per coerenza sintattica usiamo

`punto.x` ***invece di*** `punto->x`

.

Class String

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

Consultate il "javadoc"

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);
```

Class String

Constructor Summary

String()
Initializes a newly created String object so that it represents an empty character sequence.

String(byte[] bytes)
Constructs a new String by decoding the specified array of bytes using the platform's default charset.

String(byte[] ascii, int hibyte)
Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a charset name or that use the platform's default charset.*

String(byte[] bytes, int offset, int length)
Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

String(byte[] ascii, int hibyte, int offset, int count)
Deprecated. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a charset name or that use the platform's default charset.*

String(byte[] bytes, int offset, int length, String charsetName)
Constructs a new String by decoding the specified subarray of bytes using the specified charset.

String(byte[] bytes, String charsetName)
Constructs a new String by decoding the specified array of bytes using the specified charset.

String(char[] value)
Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

String(char[] value, int offset, int count)
Allocates a new String that contains characters from a subarray of the character array argument.

String(String original)
Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

String(StringBuffer buffer)
Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

Class String

Method Summary	
char	**charAt**(int index) Returns the character at the specified index.
int	**compareTo**(Object o) Compares this String to another Object.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contentEquals**(StringBuffer sb) Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
static String	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.
static String	**copyValueOf**(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
boolean	**endsWith**(String suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(Object anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(String anotherString) Compares this String to another String, ignoring case considerations.
byte[]	**getBytes**() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
void	**getBytes**(int srcBegin, int srcEnd, byte[] dst, int dstBegin) **Deprecated.** *This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the the getBytes() method, which uses the platform's default charset.*

Class String

Method Detail

length

```
public int length()
```

Returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.

Specified by:

`length` in interface `CharSequence`

Returns:

the length of the sequence of characters represented by this object.

---

charAt

```
public char charAt(int index)
```

Returns the character at the specified index. An index ranges from `0` to `length() - 1`. The first character of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

Specified by:

`charAt` in interface `CharSequence`

Parameters:

`index` - the index of the character.

Returns:

the character at the specified index of this string. The first character is at index `0`.

Throws:

`IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of this string.

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

Vi ricordate del C++ ?

```
int * z;

Z=new int[10]
```

Questo è C++

```
int z [];

z=new int[10]
```

Questo è C++,
ma è anche Java!

Array di oggetti

```
Person ** p;
    p = new Person * [10];
    p[0]=new Person("Marco");
    p[7]=new Person("Pluto");
```

```
Person   [] p;
    p = new Person * [10];
    p[0]=new Person("Marco");
    p[7]=new Person("Pluto");
```

Array di oggetti: definizione

A

```
Person[ ]  person;
person = new Person[20];

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

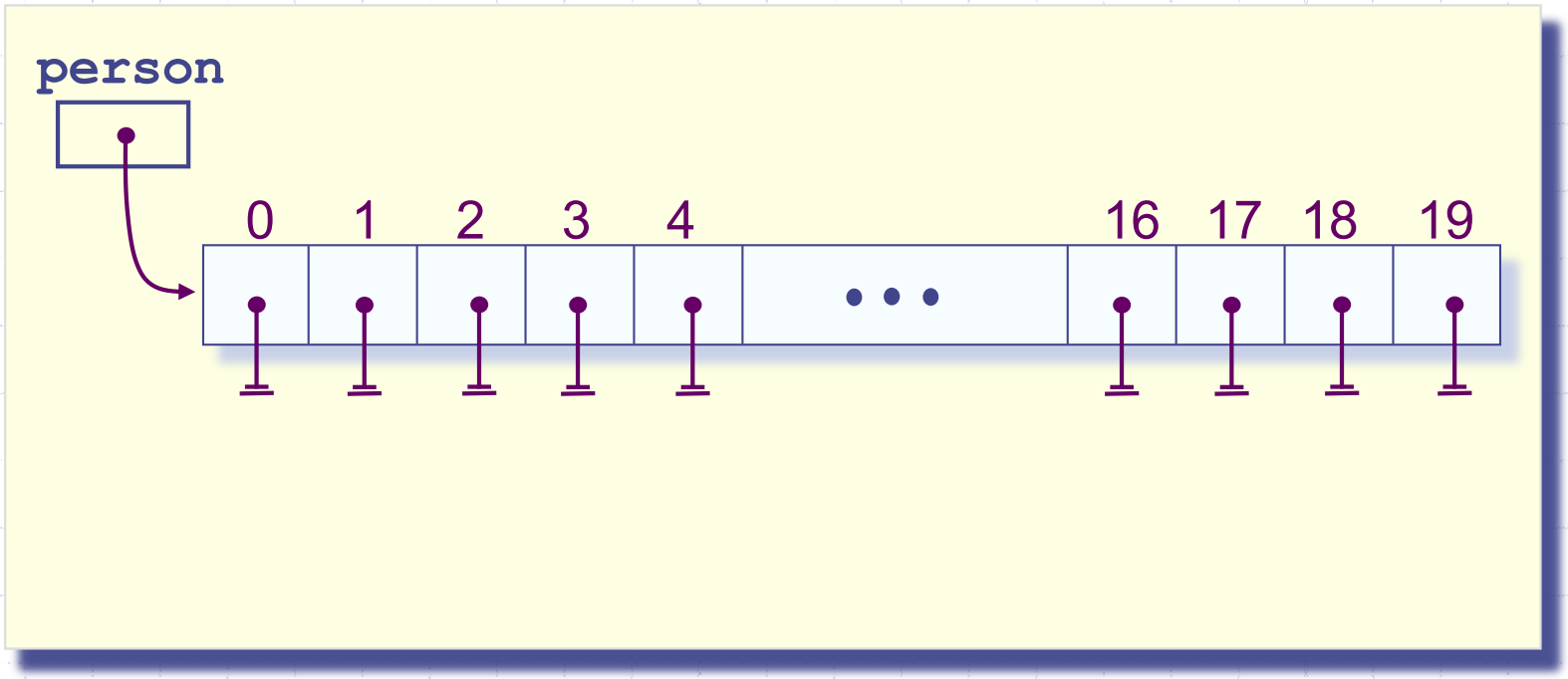
È definita solo la variabile **`person`**. L'array vero e proprio non esiste

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

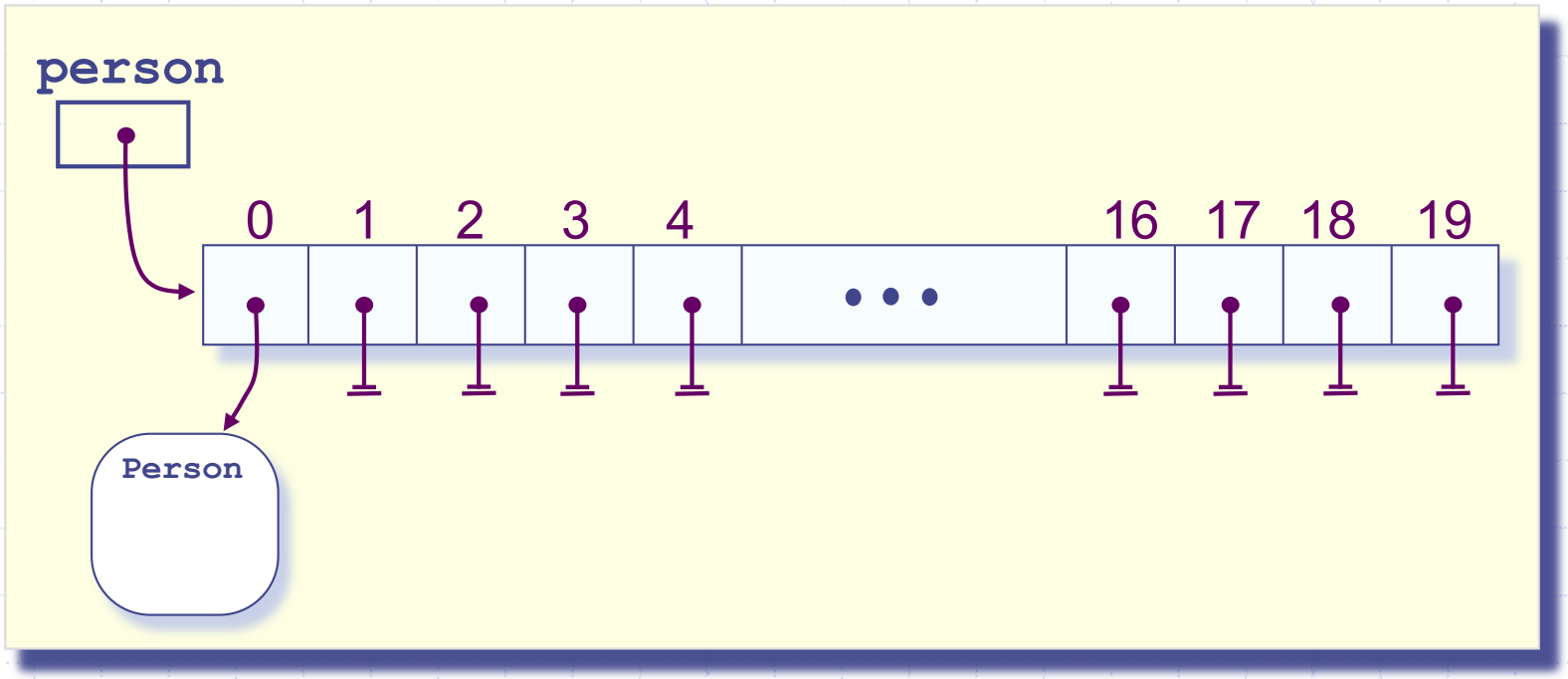


Array di oggetti: definizione

C

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

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

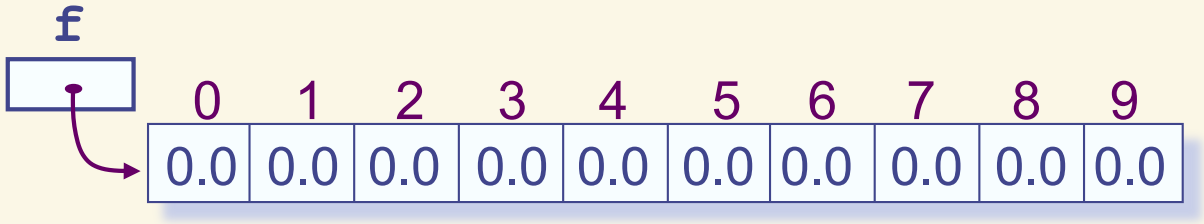


Array di oggetti vs. array di tipi base

L'istruzione:

```
float f[] = new float[10];
```

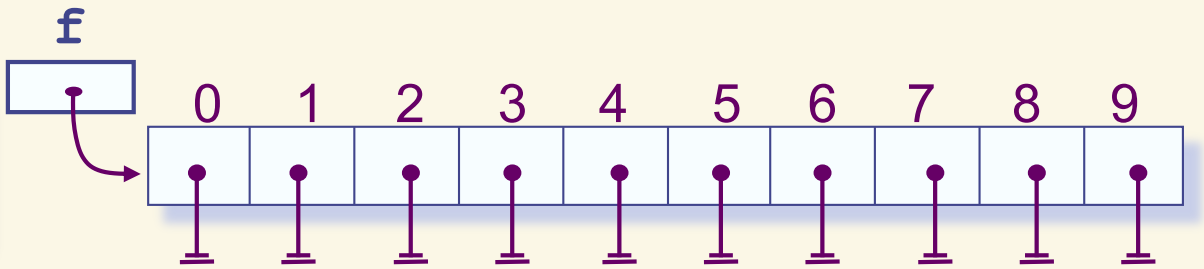
crea un oggetto di tipo array di **`float`** e alloca spazio per 10 **`float`**



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

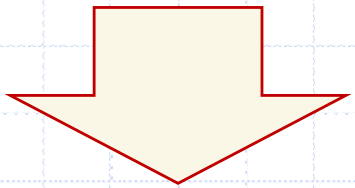
Using `System.arrayCopy()`

```
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 **`srt_position`**, nell'array **`dst`**, a partire dall'elemento con indice **`dst_position`**

`cresci`: implementazione alternativa

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



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

`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!

Lasciamolo un mistero per ora, ne discuteremo in dettaglio!

Una classe in C++

```
class Person {
public:
    Person(char * name);
    Person(const Person& orig);
   ~Person();
private:
    char * name;
};

Person::Person(char * name) {
    this->name=name;
}
```

```
public class Person {
    private String name;
    Person(String name) {
            this.name=name;
    }
}
```

Java

Copy constructor

distruttore

Copy constructor: usato per fare una copia di un oggetto dato
Chiamato implicitamente nelle assegnazioni
Person p, q; … p=q