

Il Linguaggio Java

Classi ed oggetti



```
class BankAccount {
```

```
    public BankAccount() {  
        balance = 0;  
    }
```

```
    public void deposit(double amount) {  
        balance += amount;  
    }
```

```
    public void withdraw(double amount) {  
        balance -= amount;  
    }
```

```
    public double getBalance() {  
        return balance;  
    }
```

```
    private double balance;
```

```
}
```

Costruttori

(inizializzazione dell'oggetto)

Operazioni o metodi

(operazione sull'oggetto; comportamento dell'oggetto)

Campi / Variabili istanza

(stato dell'oggetto)



```
BankAccount mio = new BankAccount();
```

- Un ***oggetto di tipo classe*** è un'istanza di una classe
- L'istanza di una classe è creata esplicitamente per mezzo di ***new***
- Una ***variabile di tipo classe*** contiene un *riferimento* ad un oggetto
- Tramite la variabile oggetto è possibile invocare metodi dell'oggetto

```
mio.deposit(1000);
```

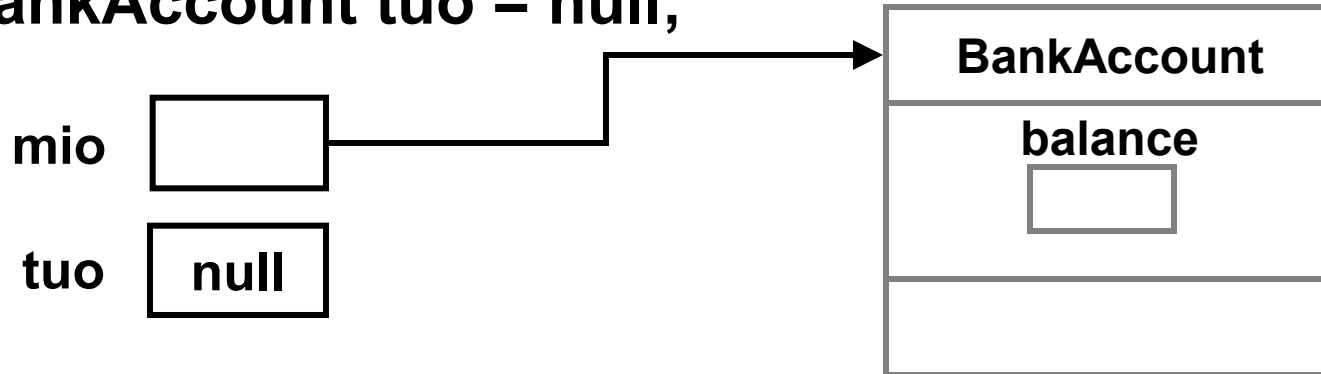
Esempio



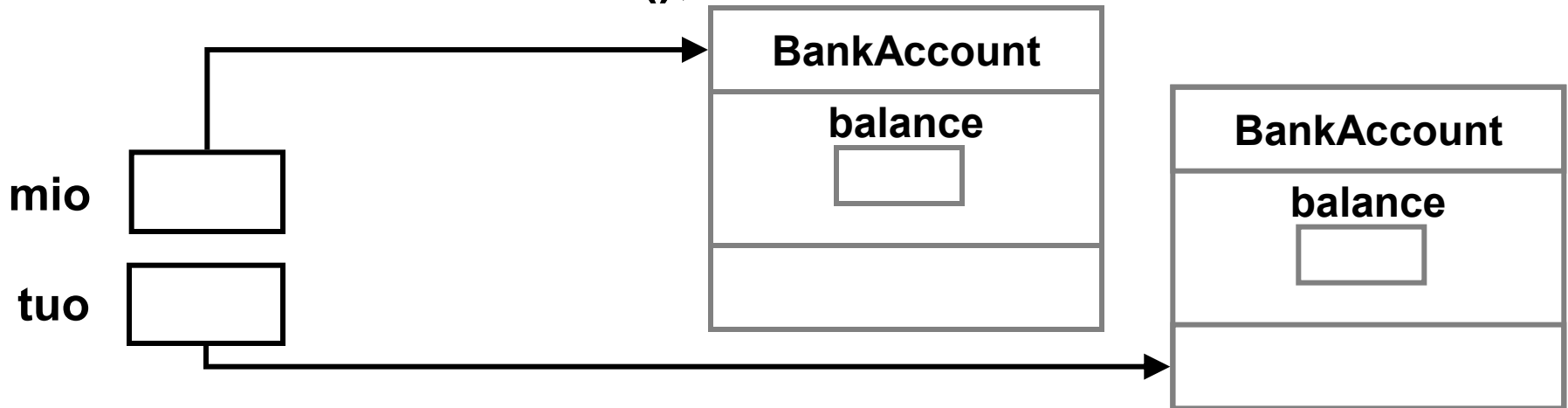
Oggetti e riferimenti

```
BankAccount mio = new BankAccount();
```

```
BankAccount tuo = null;
```



```
tuo = new BankAccount();
```





Il costruttore

- ha lo stesso nome della classe
- può avere zero o più parametri
- non ha valore di ritorno
- può essere chiamato solo insieme a **new**
(*espressione di creazione di una istanza di classe*)



Overloading dei costruttori

- Una classe può avere uno o più costruttori che si differenziano per i parametri (**overloading**)
- Se una classe non prevede esplicitamente alcun costruttore allora opera implicitamente il **costruttore di default**
 - Nessun parametro
 - I campi sono inizializzati a zero o **false** nel caso dei **boolean**
- Se una classe sono definiti i costruttori, allora quello di default non e' più disponibile.

Esempio

Esempio



Parametri formali ed attuali

```
public void deposit (double amount) {  
    balance = amount;  
}
```

Parametro formale

```
mio.deposit(x + 100);
```

Parametro attuale

Parametri impliciti ed espliciti



```
public void deposit (double amount) {  
    balance = amount;  
}
```

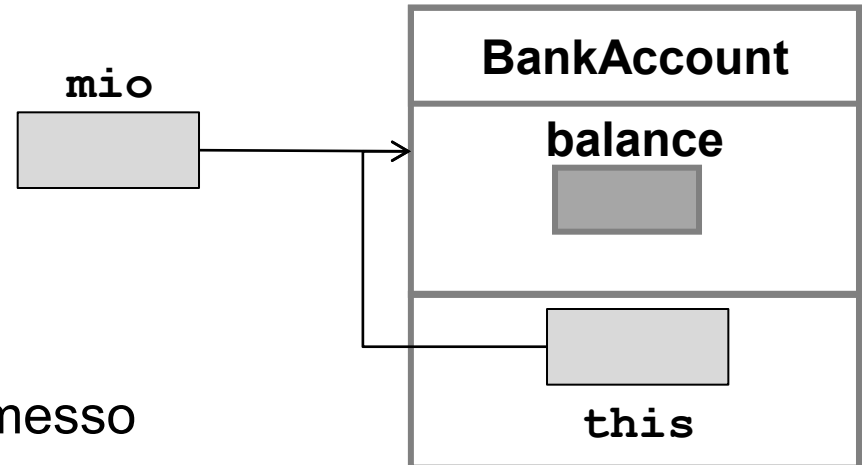
Ciascun metodo ha due classi di parametri:

- zero o più parametri **espliciti**
- un parametro **implicito** costituito dal riferimento all'oggetto di cui è stato invocato il metodo;
 - il parametro implicito è accessibile tramite la parola chiave **this**
 - la parola chiave **this** può essere usata solo all'interno del corpo di un metodo;



Il riferimento **this**

- La parola chiave **this**, rappresenta il riferimento implicito all'oggetto
 - `BankAccount mio = new BankAccount();`
 - **this** puo' essere considerato un membro della classe
 - Variabili istanza, costruttori e metodi vengono individuati tramite **this**
 - `this.balance`
 - `this.deposit()`
 - In molti casi **this** puo' essere omissso
 - In alcuni casi e' obbligatorio
 - se l'oggetto implicito e' parametro attuale o valore di ritorno;
 - se un metodo deve usare una variabile istanza che ha lo stesso nome di un parametro





Il riferimento this

```
public BankAccount(double initialBalance) {  
    balance = initialBalance;  
}
```

1

```
public BankAccount(double balance) {  
    this.balance = balance;  
}
```

2

La soluzione 1 e' da preferirsi per maggiore leggibilità



Il riferimento `this`

- Permette di invocare un costruttore da un altro costruttore
 - `this(argumentList);`
Viene invocato il costruttore specificato da `argumentList` (overloading)
 - Evita la duplicazione del codice
 - Limitazioni
 - si può chiamare un solo costruttore con questo metodo;
 - la chiamata al costruttore deve essere **la prima istruzione** del costruttore.

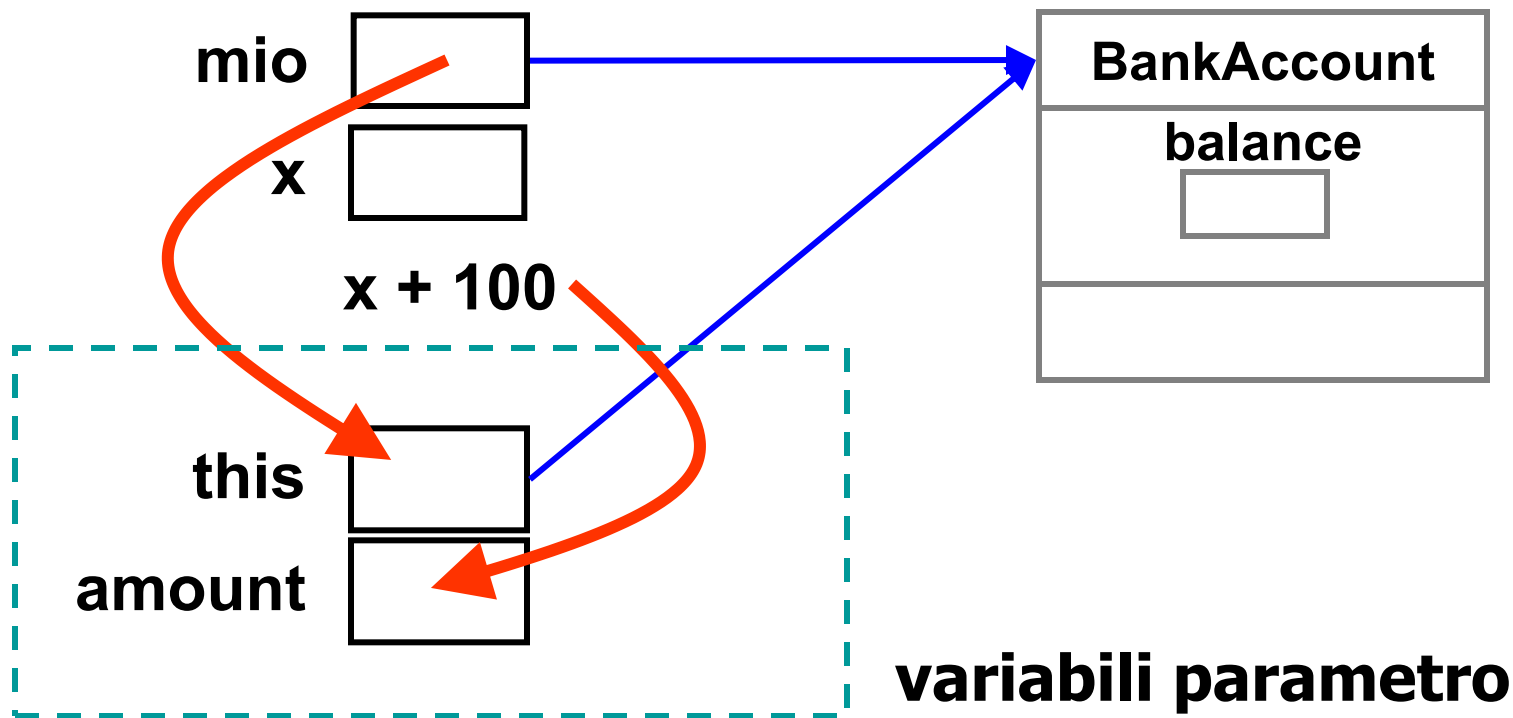


- Quando viene invocato un metodo (costruttore)
 - vengono create le **variabili parametro**, una per ogni parametro formale esplicito ed ognuna del tipo dichiarato;
 - ciascuna variabile parametro viene inizializzata con il valore dell'espressione che costituisce il corrispondente parametro attuale.
- Lo **scope** di una variabile parametro di un metodo (costruttore) è l'intero corpo del metodo (costruttore)



Trasmissione dei parametri per valore

mio.deposit (x+100);





Modifica dei parametri

- La trasmissione dei parametri e' **per valore**
- Un metodo **non può** modificare un parametro attuale di tipo primitivo
- Un metodo **non può** modificare un parametro di tipo riferimento
- Un metodo **può** modificare un oggetto riferito da un parametro

Modifica dei parametri



```
class BankAccount {
    public void deposit(double amount) {
        balance += amount; }

    public void withdraw(double amount) {
        balance -= amount; }

    public double getBalance() {
        return balance; }

    public void transfer(BankAccount other, double amount){
        balance -= amount;
        other.deposit(amount); //other.balance += amount;
    }

    private double balance;
}
```

il metodo **transfer** trasferisce la somma **amount** da **questo** conto bancario al conto **other**

Modifica dei parametri

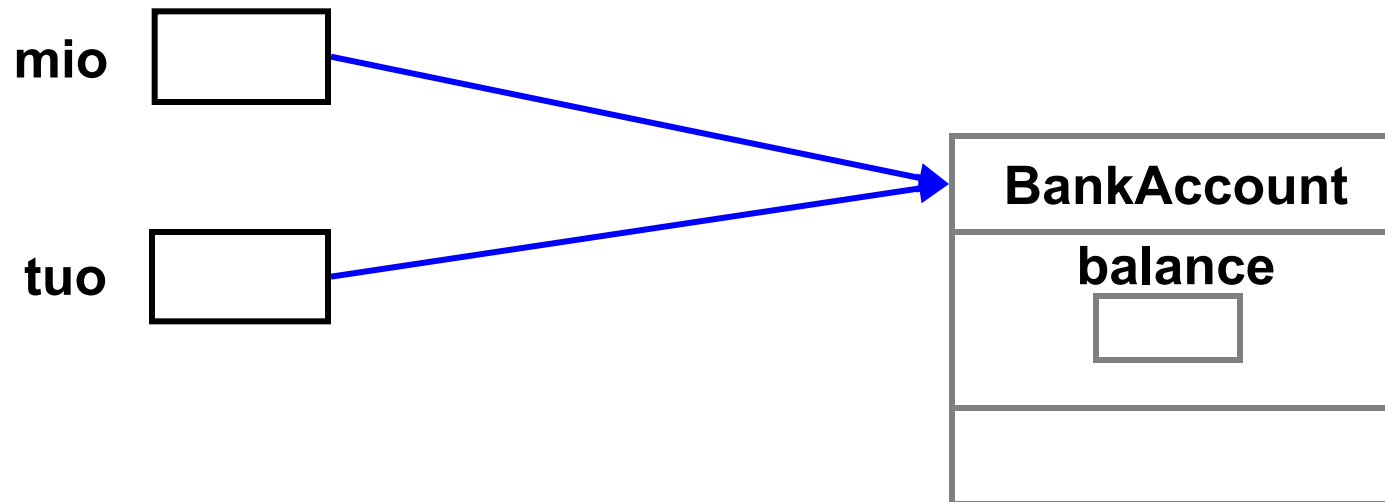


```
public void scambia (Tipo a, Tipo b) {  
    Tipo appoggio;  
    appoggio = a;  
    a = b;  
    b = appoggio;  
}  
...  
scambia(x, y); // nessuno scambio  
...
```

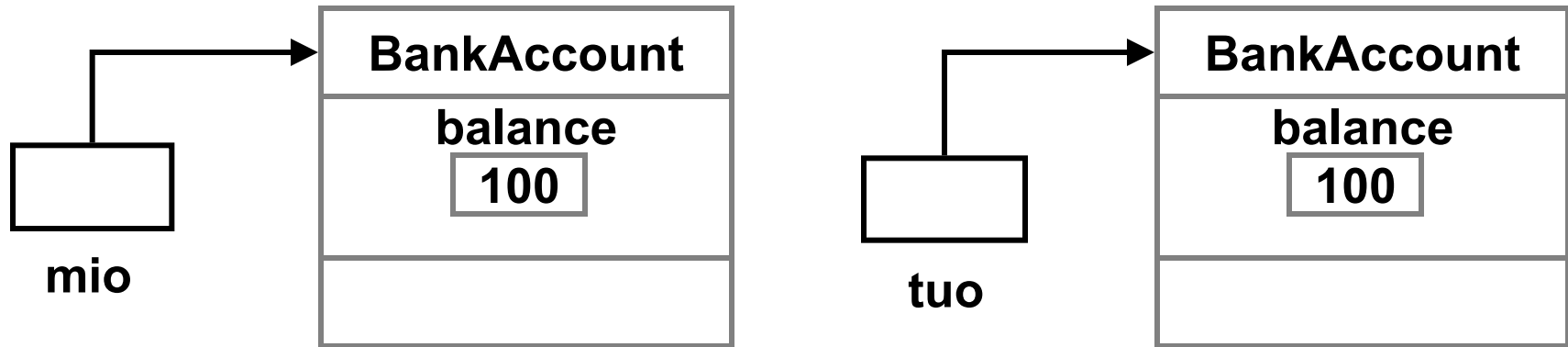



Copia di oggetti

```
BankAccount mio = new BankAccount();  
BankAccount tuo;  
tuo = mio;
```



Confronto di oggetti



```
if ( mio == tuo )  
    System.out.println("sono lo stesso oggetto);
```



Metodi accessori e modificatori

- Metodo **accessore** ritorna informazioni sull'oggetto (riferito da **this**) ma non lo modifica
- Metodo **modificatore** modifica lo stato dell'oggetto (riferito da **this**)
- Un metodo (accessore o modificatore) ha un **effetto collaterale** se ha un qualunque tipo di comportamento osservabile che esula dall'oggetto (riferito da **this**)
 - modifica di un parametro esplicito
 - stampa di messaggi
 - modifica di variabili di classe (variabile statica)



Metodi statici (**static**)

- Un metodo viene detto **statico**, o **metodo di classe**, quando viene dichiarato con il modificatore di accesso **static**
- Un metodo statico non ha parametro implicito **this** per cui *non* opererà su una particolare istanza della classe
 - I metodi non-statici sono detti **metodi di istanza** proprio perché operano su una particolare istanza della classe
- Per riferire un metodo statico si utilizza il **nome qualificato**

Campi statici (**static**)

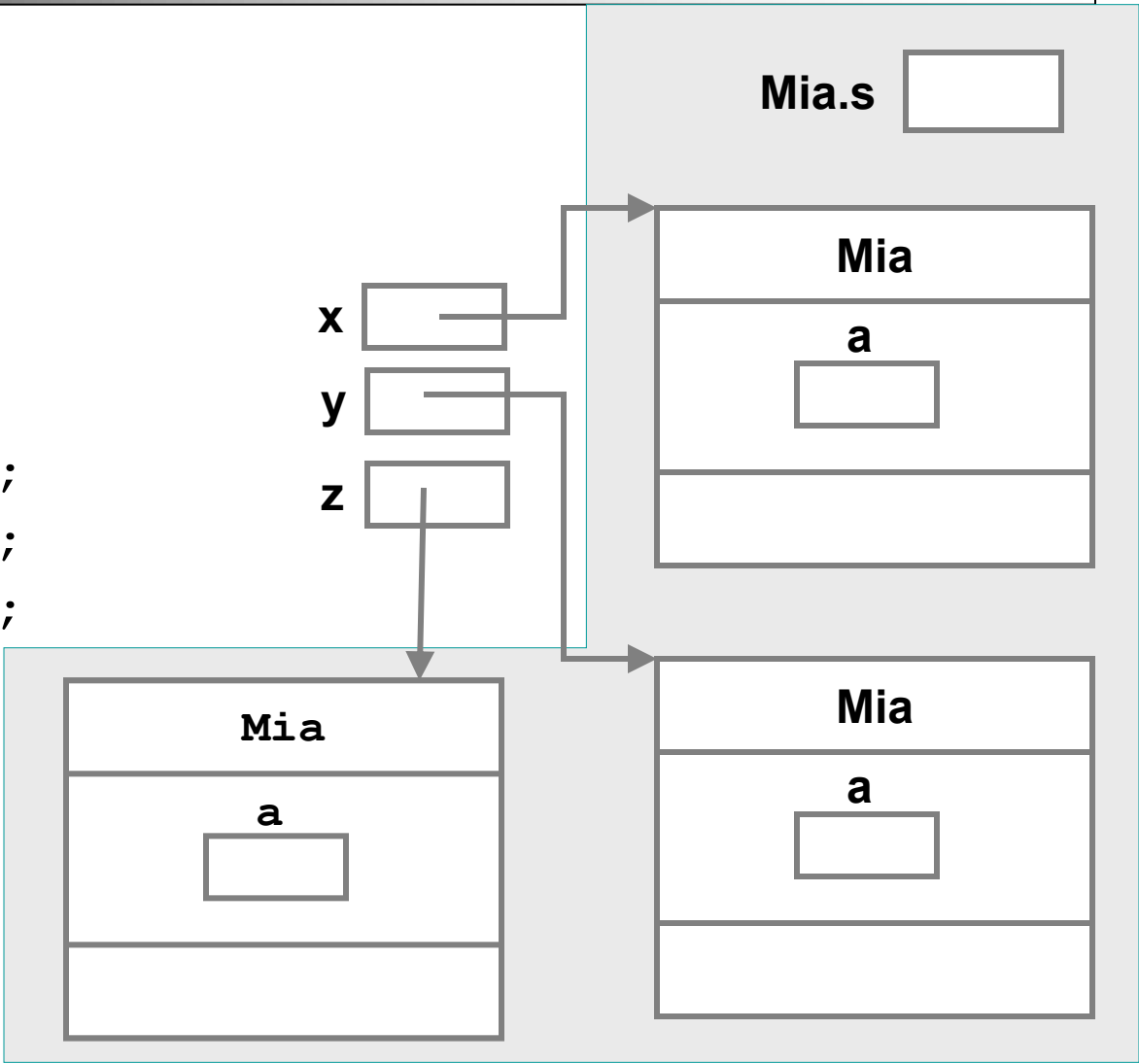


- Un campo viene detto *static*, o *variabile di classe*, quando viene dichiarato con il modificatore di accesso **static**
- Di un campo **static** esiste una sola incarnazione indipendentemente da quante istanze (*anche zero*) della classe possono essere state create
 - Un campo non-**static** è detto *variabile istanza* e ne esiste un'istanza per ciascun oggetto

Campi statici (static) (II)



```
classe Mia {  
  ...  
  static int s;  
  int a;  
  ...  
}  
Mia x = new Mia();  
Mia y = new Mia();  
Mia z = new Mia();
```





Inizializzazione delle variabili locali

```
classe MiaClasse {
```

```
...
```

```
int f( /* ... */ ) {
```

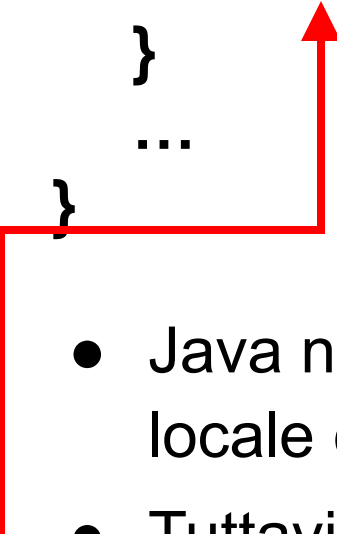
```
    int i; // variabile locale
```

```
    i++; // compile-time error
```

```
}
```

```
...
```

```
}
```

- 
- Java non garantisce un valore iniziale ad una variabile locale che deve quindi essere inizializzata esplicitamente
 - Tuttavia, il compilatore genera un errore se una variabile locale viene usata prima di essere stata inizializzata



- INIZIALIZZAZIONE AUTOMATICA
- INIZIALIZZAZIONE ESPLICITA
- INIZIALIZZAZIONE TRAMITE IL COSTRUTTORE



Inizializzazione Automatica

Java garantisce un valore iniziale di *default* per ciascun campo (variabile istanza):

- zero per i campi primitivi; **false** per i **boolean**
- **null** per i campi riferimento

```
classe MiaClasse {
```

```
...
```

```
int i;           // 0
```

```
boolean b;      // false
```

```
char c;         // \u0020 (spazio)
```

```
double d;       // 0.0
```

```
AltraClasse a;  // null
```

```
...
```

```
}
```

Il costruttore di default imposta le variabili istanza ai loro valori di default



Inizializzazione esplicita

È possibile assegnare un valore iniziale specifico ad un campo per mezzo di un assegnamento nel punto in cui il campo è dichiarato

```
class MiaClasse {  
    ...  
    int i = 1;  
    int h = f(3);  
    int k = g(h);  
    // int k = h(b); b non inizializzato  
    boolean b = true;  
    char c = 'z';  
    double d = 2.3;  
    AltraClasse a = new AltraClasse();  
    ...  
}
```

ordine dell'inizializzazione
↓

Inizializzazione mediante il costruttore



```
classe MiaClasse {
```

```
    ...
```

```
    int i; // 0
```

```
    double d = 2.3;
```

```
    public MiaClasse(int a, double b) {
```

```
        ...
```

```
        i = a;
```

```
        ...
```

```
        d = b;
```

```
    }
```

```
    ...
```

```
}
```

- Maggiore flessibilità
- La inizializzazione automatica o con inizializzatore avviene **prima** che venga invocato qualunque metodo o costruttore



Inizializzazione

```
Class Tua {
    int a = 1;
    int b = a++;
    Tua (int n) {
        a = n;
    }
    ...
}
```

```
Class Prova {
    public static void main(String[] args) {
        ...
        Tua t1 = new Tua(5); // b = 2; a = 5
        Tua t2 = new Tua(10); // b = 2; a = 10
    }
}
```



- **INIZIALIZZAZIONE AUTOMATICA**

Una variabile statica viene automaticamente inizializzata

- su **0**, per i tipi numerici, **false** per il tipo **boolean**
- **null** per i tipi riferimento

- **INIZIALIZZATORE ESPLICITO**

```
static int a = 1;
```

- **BLOCCO DI INIZIALIZZAZIONE STATICA**

```
static int a;  
static {  
    a = 1;  
}
```

Non ha senso usare un costruttore per inizializzare una variabile statica



Metodi privati e campi costanti

```
class BankAccount {
```

```
    public BankAccount() {this(0);}

```

```
    public BankAccount(double initialBalance) {
        accountNumber = getNewAccountNumber();
        balance = initialBalance; }

```

```
    private int getNewAccountNumber() {
        return newAccountNumber++;
}
```

← Un METODO PRIVATO può essere chiamato solo dagli altri metodi della stessa classe

```
    // deposit, withdraw, getBalance

```

```
    public int getAccountNumber() {
        return accountNumber; }

```

```
    private double balance;

```

```
    private final int accountNumber;
```

```
    private static int newAccountNumber = 1;

```

← Il CAMPO COSTANTE viene inizializzato nel costruttore al momento della creazione dell'oggetto

```
}
```



Metodi privati e campi costanti

```
class BankAccount {  
    public BankAccount() {this(0);}  
    public BankAccount(double initialBalance) {  
        balance = initialBalance; }  
    private int static getNewAccountNumber() { ← rendendo static il  
        return ++newAccountNumber;          metodo  
    }  
    // deposit, withdraw, getBalance  
    public int getAccountNumber() {  
        return accountNumber; }  
    private double balance;  
    private final int accountNumber = getNewAccountNumber();  
    private static int newAccountNumber = 1;  
}
```

Tempo di vita delle variabili



- **Variabile istanza:** ha un tempo di vita pari a quello dell'oggetto a cui appartiene
- **Variabile statica** (di classe): viene creata quando si carica la classe in cui è dichiarata; cessa di esistere quando la classe viene scaricata
- **Variabile locale:** viene creata quando il flusso entra nel blocco in cui è immediatamente contenuta; cessa di esistere quando il flusso del controllo lascia il blocco
- **Variabile parametro:** viene creata quando inizia l'esecuzione di un metodo (costruttore); cessa di esistere quando l'esecuzione del metodo (costruttore) termina



Inizializzazione

- **Variabile istanza:** inizializzazione automatica su zero; inizializzatore esplicito; costruttore
- **Variabile statica** (di classe): inizializzazione automatica su zero; inizializzatore o blocco **static**
- **Variabile locale:** dichiarazione con inizializzazione; assegnamento
- **Variabile parametro:** copia del parametro effettivo



- **Variabile istanza e variabile statica:** sono tipicamente dichiarate con il modificatore di accesso **private** che ne restringe l'ambito solamente ai metodi della classe
- **Variabile locale:** l'ambito si estende dal punto in cui la variabile è definita fino alla fine del blocco
- **Variabile parametro:** l'ambito comprende tutto il corpo del metodo/costruttore



Shadowing

- Due variabili con lo stesso nome possono avere ambiti sovrapposti.
- In tal caso, una delle variabili mette in ombra (**shadowing**) l'altra
- ESEMPIO I: variabile istanza e (variabile) parametro

```
public class BankAccount {  
    private double balance;  
    public BankAccount(double balance) {  
        this.balance = balance;  
    }  
    // gli altri metodi  
}
```

il parametro mette in ombra la variabile istanza per tutto l'ambito del parametro



Shadowing

- La dichiarazione di una variabile (istanza/statica, locale o parametro) con un certo nome mette in ombra le variabili (istanza/statica, locale e parametro) con lo stesso nome che sono visibili nel punto in cui appare la dichiarazione per tutto l'ambito della variabile.

```
scope di  
a(1) [ int a; // 1  
      .  
      .  
      .  
      int a; // 2 ] scope di  
      .  
      .  
      .  
a(2)
```

a (2) mette in ombra a (1) in tutto il suo scope



Classi su file distinti (I)

```
public class BankAccount {  
    // membri  
}
```

BankAccount.java

```
public class BankAccountDriver {  
    public static void main(String args[]) {  
        BankAccount mio = new BankAccount();  
        mio.deposit(25.8);  
    }  
}
```

BankAccountDriver.java



Classi su file distinti (II)

- **javac BankAccountDriver.java**

- Il compilatore compila **BankAccountDriver.java** e rileva la presenza della classe **BankAccount**; quindi
- Il compilatore cerca **BankAccount.class**;
 - se non lo trova cerca **BankAccount.java** e lo compila
 - se lo trova ma risulta più vecchio di **BankAccount.java** allora questo file viene ricompilato

- **java BankAccount**



Classi su file distinti (III)

BankAccount.java

```
public class BankAccount {  
    // membri  
    public static void main(String args[]) {  
        // istruzioni  
    }  
}
```

Applicazione.java

```
public class Applicazione {  
    public static void main(String args[]) {  
        BankAccount mio = new BankAccount();  
        mio.deposit(25.8);  
    }  
}
```

BankAccount può essere testata separatamente da **Applicazione**



- **EFFICIENZA**

- Il numero, l'occupazione di memoria ed il tempo di vita degli oggetti è determinato in fase di compilazione
- Gli oggetti sono allocati sullo stack o nella memoria statica
- La distruzione degli oggetti è controllata attraverso le regole di visibilità
- Gli oggetti sono “poco complicati”

- **FLESSIBILITÀ**

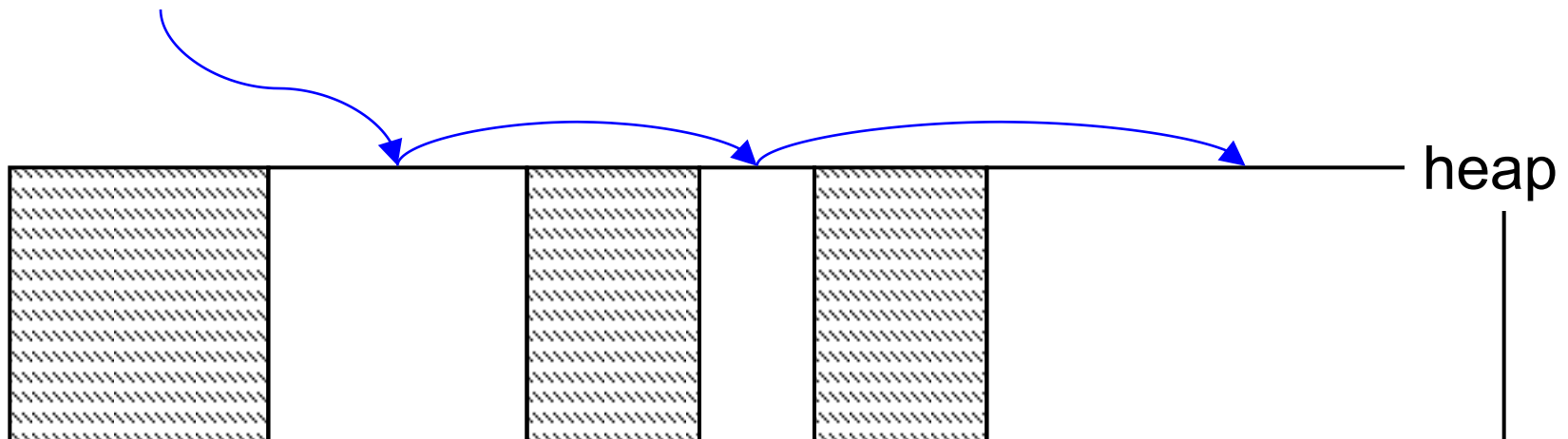
- Gli oggetti sono creati e distrutti dinamicamente
- Gli oggetti sono allocati nello heap
- Gli oggetti sono “complicati”



Gestione dello heap alla C++

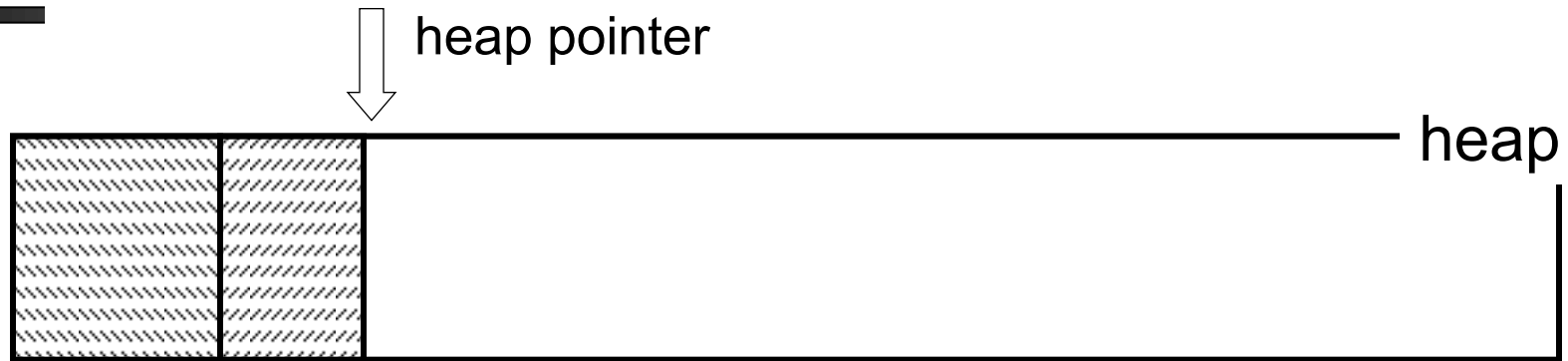
- Tempo di allocazione sullo heap molto maggiore del tempo di allocazione sullo stack
- Deallocazione efficiente ma lasciata al programmatore
- Memory leakage

memoria libera





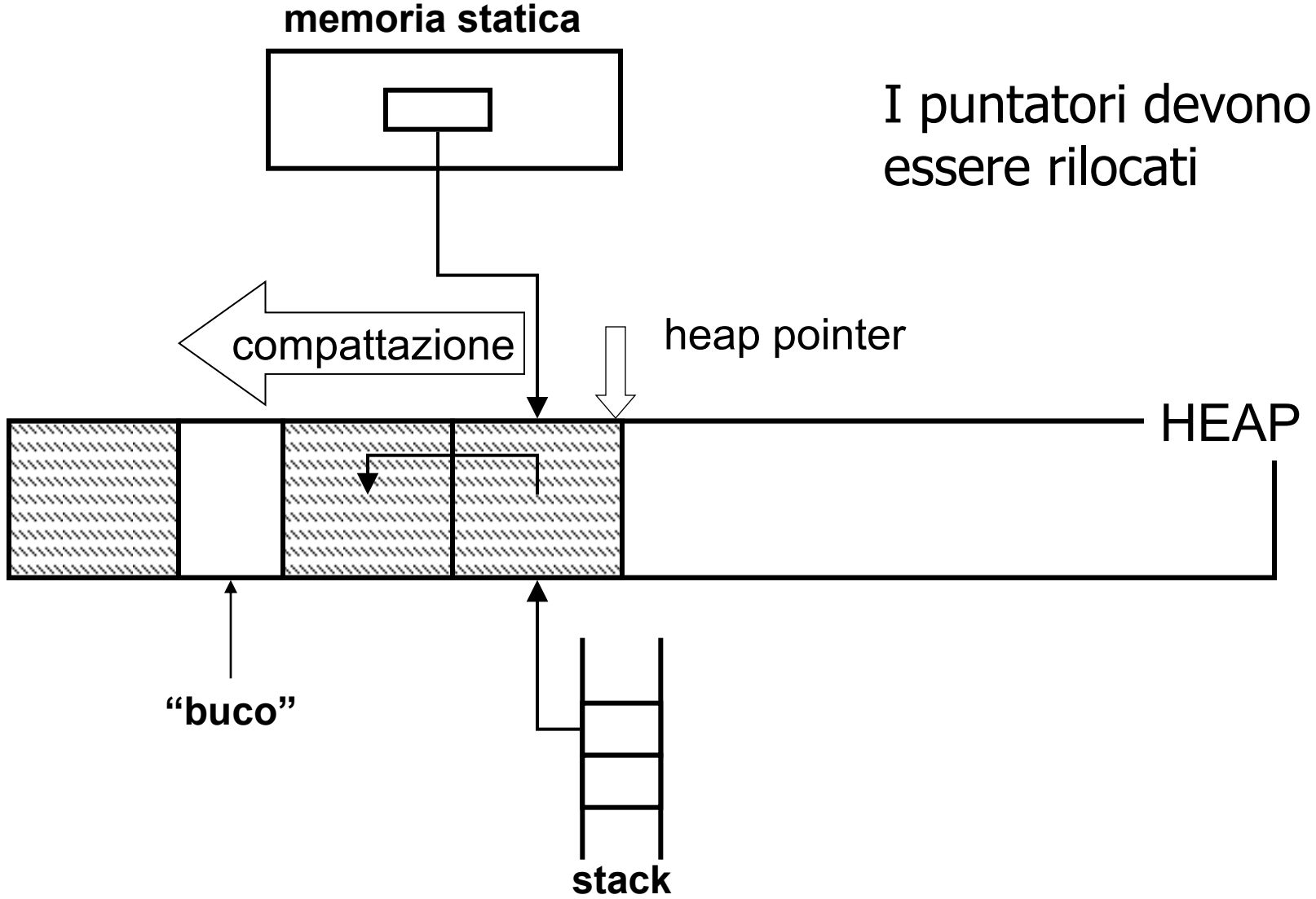
Gestione dello heap



- Allocazione efficiente
- La deallocazione è affidata al supporto run-time del linguaggio: il **Garbage Collector**
- GC si “accorge” quando un oggetto non è più utilizzato e lo “raccoglie”, “reclamando” l’area di memoria riservata all’oggetto
- Il GC gira quando lo ritiene opportuno; mentre il GC gira, il programma è "sospeso"



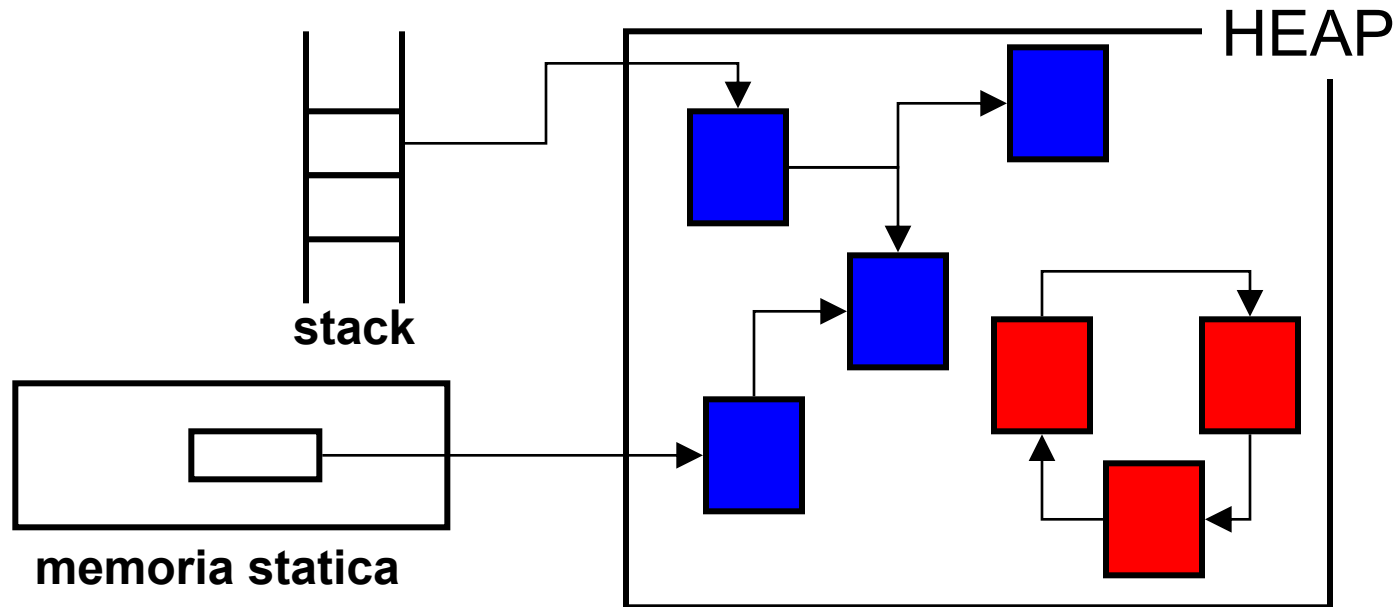
Compattazione





Determinazione oggetti attivi

Il GC raccoglie gli oggetti che sono **inattivi**

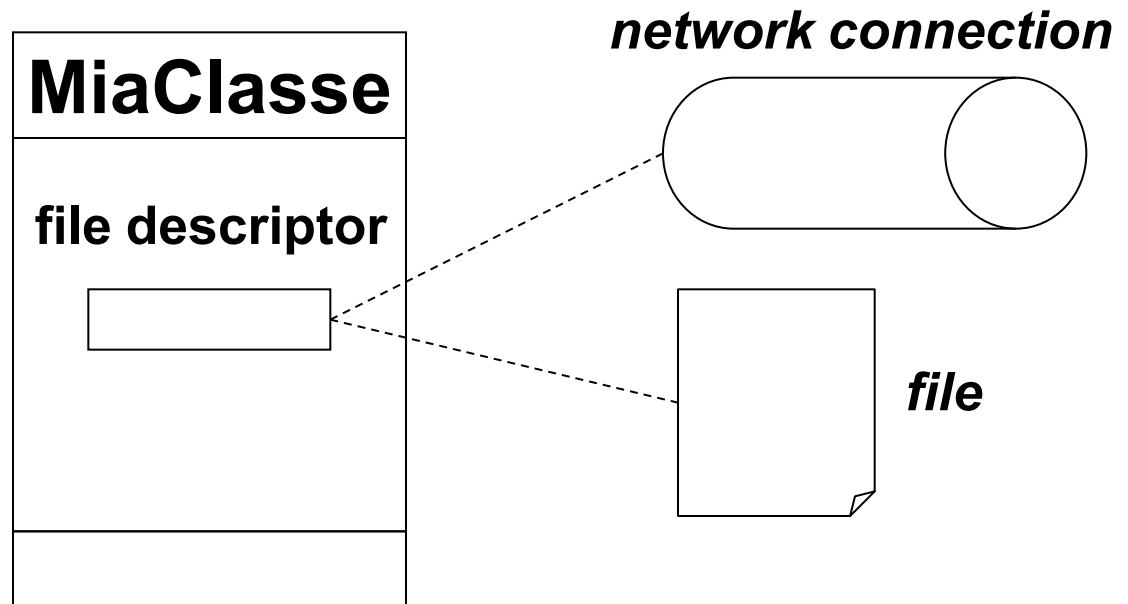


Un oggetto è **attivo** se è raggiungibile a partire da un riferimento memorizzato sullo stack o in memoria statica



Il metodo finalize

Un oggetto, oltre alla memoria può utilizzare altre risorse



Quando il GC raccoglie l'oggetto, tali risorse devono essere rilasciate (**cleanup**)

il GC rilascia la memoria... chi rilascia queste risorse?



Il processo di finalizzazione

- quando un oggetto viene raccolto, GC invoca il metodo **finalize** per eseguire il cleanup
- Il metodo **finalize** può essere (ri)definito in ogni classe
- Non è consigliabile utilizzare il metodo **finalize** per gestire risorse scarse
non si può sapere se e quando GC girerà

Overloading



- Overloading dei costruttori ed overloading dei metodi seguono le stesse regole
- *Signature* di un metodo è costituita dal **nome del metodo** e dal **numero e tipo dei parametri** del metodo
 - Una classe non può avere due metodi con la stessa signature
- Due metodi di una classe sono *sovrapposti* se hanno lo stesso nome ma signature differenti



Invocazione di un metodo (I)

- La chiamata di un metodo deve
 - determinare la **signature** del metodo
 - determinare i metodi **applicabili** ed **accessibili**
 - tra questi selezionare quello più **specifico**



Invocazione di un metodo (II)

- Un metodo è **applicabile** se e solo se
 - numero degli parametri = numero degli argomenti
 - il tipo di ciascun argomento **può essere convertito** nel tipo del corrispondente parametro
(*conversione nella chiamata*)
- L'**accessibilità** di un metodo è determinata dai **modificatori d'accesso**
- Un metodo è più **specifico** di un altro se ogni chiamata gestita dal primo può essere gestita anche dal secondo senza errori di compilazione



ESEMPIO

```
class A {
    public static void p() { System.out.print("p() "); p(1); }
    private static void p(int i) { System.out.println("p(int) ");}
}

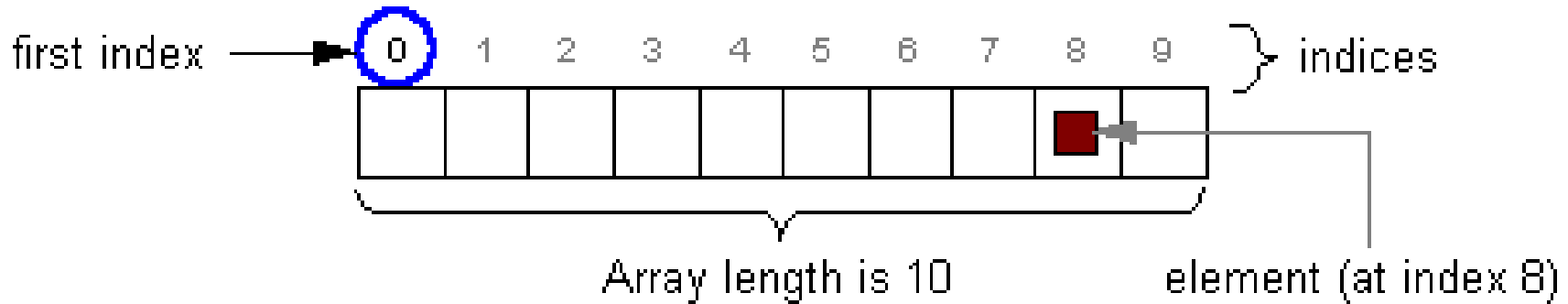
public class MethodOverloading {
    public static void q(int i) {System.out.println("q(int)");}
    public static void q(long n) {System.out.println("q(long)");}
    public static void main(String[] args) {
        A.p(); // p() p(int)
        // A.p(1); compile-time error – non accessibile
        // q(2.5); compile-time error – non applicabile
        short s = 1;
        q(s); // q(int)
        q(1); // q(int)
    }
}
```

Array

Classi ed oggetti



Il modello



- Un array contiene valori dello stesso tipo (primitivo o riferimento)
- La lunghezza di un array è definita a run-time quando l'array viene creato
(un array può avere lunghezza pari a zero)
- Un elemento dell'array viene acceduto specificando la sua posizione nell'array tramite un indice



Dichiarazione e creazione di un array

Dichiarazione

- `int a[];`
- `int[] a; // stile preferito`
- *`type[] nome;`*

La dichiarazione di una variabile array non alloca memoria per gli elementi dell'array e non specifica la dimensione dell'array

Creazione di un array

- `a = new int[100];`
- *`new elementType[arraySize]`*

Si alloca memoria per un array di *arraySize* elementi di tipo *elementType*



Accesso agli elementi (I)

- Accesso all'array tramite indice

- `a[i] = 5;`
- `a[i] = a[j+1];`

L'indice deve essere di tipo **int**

- L'operatore `[]` dà luogo a promozione unaria, perciò

- **byte**, **short** e **char** possono essere utilizzati come indici
- **long** dà luogo ad compile-time error

- Non è definita l'aritmetica dei puntatori

```
int[] a = new int[10];  
// a++; compile-time error
```



Accesso agli elementi (II)

- L'operatore `[]` garantisce il **bound checking** a run-time
 - Dato *arrayName*[*expression*], l'interprete controlla a run-time che $0 \leq \mathbf{expression} < \mathbf{arrayLength}$
 - La variabile `length`
 - La lunghezza di un array è disponibile a run-time attraverso la variabile istanza **public** e **final** di nome `length`
- ```
int[] a = new int[10];
for (int i = 0; i < a.length; i++)
 a[i]=i*i;
```



# Inizializzazione

---

- Per default, gli elementi dell'array sono inizializzati a zero
- Nella dichiarazione per mezzo di un iniziatore  
`int[] a = {2, 3, 5, 7, 11, 13};`
- Tramite un array anonimo  
`int[] a = new int[] {17, 19, 23, 29, 31, 37};`
- A programma, elemento per elemento  
`int[] a = new int[20];  
for (i = 0; i < a.length; i++)  
 a[i] = i*i;`





# Esempio

---

Definire un array di lunghezza casuale tra 1 e 20 ed inizializzare i suoi elementi con valori interi casuali compresi tra 1 e 100

```
import java.util.*;
...
Random rand = new Random();
...
int[] a;
...
a = new int[rand.nextInt(20)];
for (int i = 0; i < a.length; i++)
 a[i] = rand.nextInt(100);
...
```



# Classi wrapper

---

- La **classe wrapper** di un tipo primitivo permette di creare un oggetto che contiene un valore del tipo primitivo

**Byte, Short, Integer, Long, Float,  
Double, Character, Boolean**

- Esempio: la classe **Integer**

```
int i = 3;
```

```
Integer I = new Integer(i);
```

```
Integer J = new Integer("123");
```

```
String t = "-123";
```

```
int j = Integer.parseInt(t); // formato decimale
```

```
String s = I.toString(); // formato decimale
```



# Array di tipo riferimento

---

## Dichiarazione e creazione

- dichiarazione di una variabile array

**Integer[] m;**

- dichiarazione di una variabile array e creazione di un oggetto array

**Integer[] m = new Integer[50];**



# Array di tipo riferimento

---

## Inizializzazione

- Per default gli elementi di un array sono inizializzati al valore **null**

- Con inizializzatore

```
Integer[] m = {new Integer(1),
 new Integer(2),
 new Integer(3)};
```

- Con array anonimo

```
Integer[] m = new Integer[]{new Integer(1),
 new Integer(2),
 new Integer(3)};
```



# Array di tipo riferimento

---

**Esempio:** Definire un array di (riferimenti ad) **Integer** di lunghezza casuale tra 1 e 20 ed inizializzare ciascun elemento con un valore casuale compreso tra 1 e 100

```
import java.util.*;
...
Random rand = new Random();
Integer a[];
a = new Integer[rand.nextInt(20)];
for (int i = 0; i < a.length; i++)
 a[i] = new Integer(rand.nextInt(100));
...
```



# Utilities

---

- **Classe Arrays (java.util.\*)**
  - **static void sort(*PrimitiveType*[] a)**
  - **static int binarySearch(*PrimitiveType*[] a, *PrimitiveType* v)**
- **Classe System**
  - **static void arrayCopy(Object from, int fromIndex, Object to, int toIndex, int count)**



# Array multidimensionali

---

## Dichiarazione

- Dichiarazione di una variabile array: **int[][] a;**
- Dichiarazione di una variabile array e creazione di un oggetto array: **int[][] a = new int[50] [100];**
- Dichiarazione di una variabile array e creazione di un oggetto array ed inizializzazione con inizializzatore  
**int[][] a = {{1, 2, 3},  
                  {4, 5, 6}};**

## Accesso per indice

```
for (int i=0; i < a.length; i++)
 for(int j=0; j < a[i].length; j++)
 a[i][j] = i+j;
```



# Ragged Arrays

---

- La dichiarazione di una variabile array non specifica la dimensione dell'oggetto array riferito
- Un array multidimensionale è un array di array
- La dimensione dei sub-array può essere lasciata non specificata

```
int[][] a = new int[50][];
```

- Le righe di un array multidimensionale possono avere lunghezza differente



# ***Stringhe e caratteri***

***Classi ed oggetti***



# La classe wrapper **Character**

---

- Un oggetto di classe **Character** memorizza un carattere
- La classe **Character** fornisce metodi utili per la manipolazione dei caratteri
- Si utilizza **Character** invece di **char** quando:
  - un metodo richiede un oggetto o un tipo primitivo
  - quando il carattere deve essere collocato in una struttura dati che richiede oggetti (es.: **Vector**)
  - quando il carattere deve essere passato ad un metodo che lo modifica



# La classe **String**

---

- Classe predefinita **String**

```
String e = ""; // stringa vuota
```

```
String s = "Hello";
```

```
String t = new String("Hello");
```

```
String u = new String(s);
```

```
char[] d = {'a', 'b', 'c'};
```

```
String v = new String(d);
```

- Le stringhe sono **immutabili**
  - *Non è possibile modificare i caratteri di una stringa*
  - *Le stringhe possono essere condivise*
- Per le stringhe mutabili si usa **StringBuffer**



# I letterali di tipo **String**

---

- Un letterale è specificato tra virgolette
  - **“Hello world!”**
- Un letterale può essere utilizzato ovunque si utilizza un oggetto **String**
- Il compilatore crea un oggetto **String** per ogni letterale
  - **String s = “Hello world!”;**
  - **String s = new String(“Hello world!”);**



# Operazioni sulle stringhe

---

- Concatenazione (+)

```
String v = "Via ";
```

```
String n = "Roma ";
```

```
String i = v + n + 2;
```

- Sottostringhe

```
String s = "Hello";
```

```
String t = s.substring(0, 4); // Hell
```

- Editing

```
int n = s.length(); // 5
```

```
char c = s.charAt(4); // 'o'
```

```
String z = s.substring(0, 4) + '!'; // Hell!
```



# Operazioni sulle stringhe

---

- Confronto tra stringhe

```
String s = "pippo";
```

```
String t = "pluto";
```

```
boolean b1 = s.equals(t); //false
```

```
boolean b2 = "pippo".equals(s); // true
```

```
String u = "Pippo";
```

```
boolean b3 = s.equalsIgnoreCase(u); //true
```



# Confronto tra stringhe

---

```
String greeting = "Hello";
if (greeting == "Hello")
 // probably true
if (greeting.substring(0, 4) == "Hell")
 // probably false
```

- ***Solo le stringhe costanti sono condivise***, mentre
- le stringhe ottenute da concatenazione ( `'+'` ) oppure da `substring` non lo sono, perciò
- ***non usare*** `'=='` per confrontare due stringhe

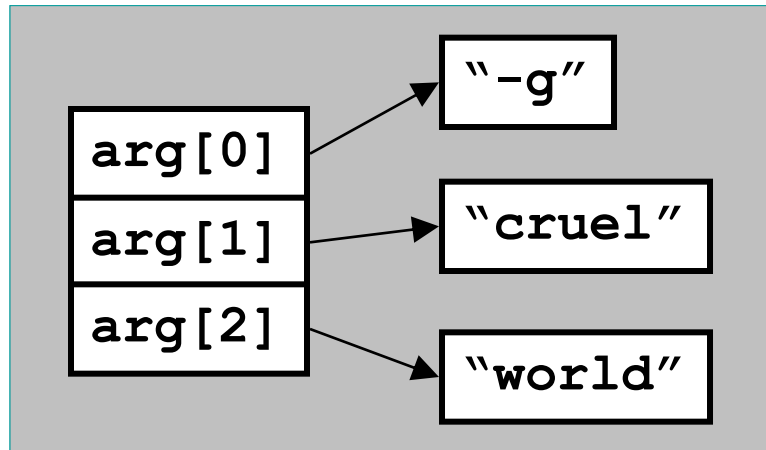


# Argomenti sulla linea di comando

---

*Il main ha come parametro un array di stringhe*

**> java ArgsDemo -g cruel world**





# Le stringhe Java e le stringhe C++

---



- In Java le stringhe sono ***first-class objects*** mentre in C++ le stringhe sono array di caratteri terminati da '\0'
- Al contrario del C++, in Java le stringhe hanno un comportamento affidabile e consistente



# Letture dell'input

```
import javax.swing.*;

public class InputTest {
 public static void main(String[] args) {
 // 1) lettura del primo input
 String name = JOptionPane.showInputDialog("What's your name?");
 // 2) lettura del secondo input
 String input = JOptionPane.showInputDialog("How old are you?");
 // 3) conversione string -> int
 int age = Integer.parseInt(input);
 // 4) display dell'output su console
 System.out.println("name: " + name);
 System.out.println("age: " + age);
 System.exit(0); //!!!
 }
}
```

