Il Linguaggio Java

Classi ed oggetti

Classe

```
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

metodi

campi

Oggetti e riferimenti

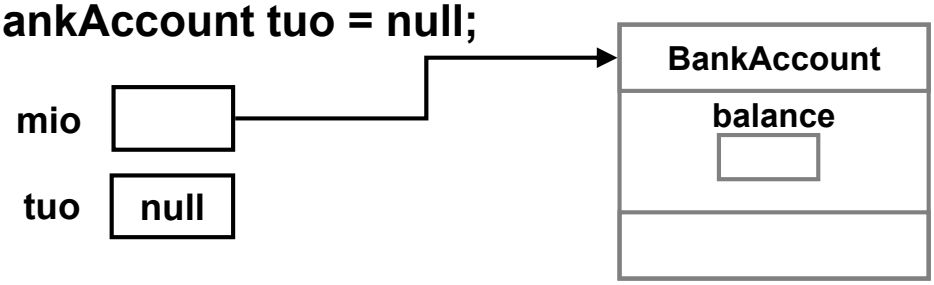
BankAccount mio = new BankAccount();

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

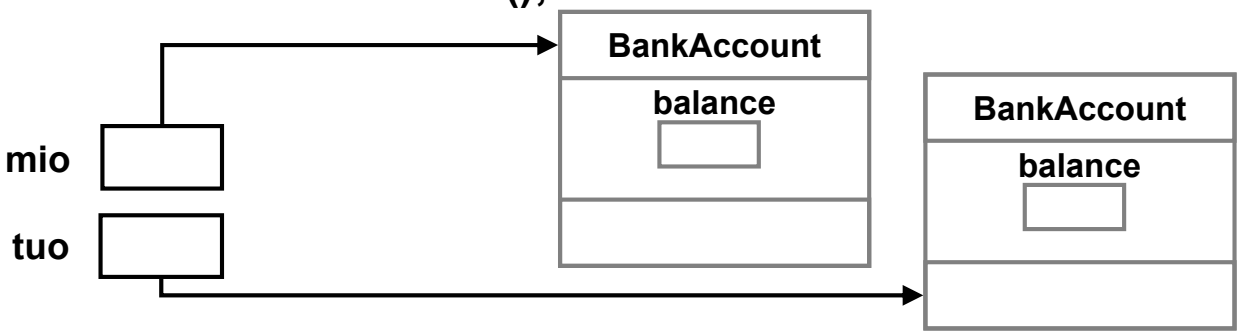
mio.deposit(1000);

Oggetti e riferimenti

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



tuo = new BankAccount();



Costruttori

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 però devono avere diversi parametri (overloading)
- Se il programmatore non specifica alcun costruttore per la classe, allora
  - il compilatore fornisce automaticamente un costruttore di default senza argomenti;
  - in tal caso le variabili istanza sono inizializzate a zero (**false** nel caso dei boolean).

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 il metodo è stato invocato;

  il parametro implicito è accessibile tramite la parola chiave **this**

Variabili parametro

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

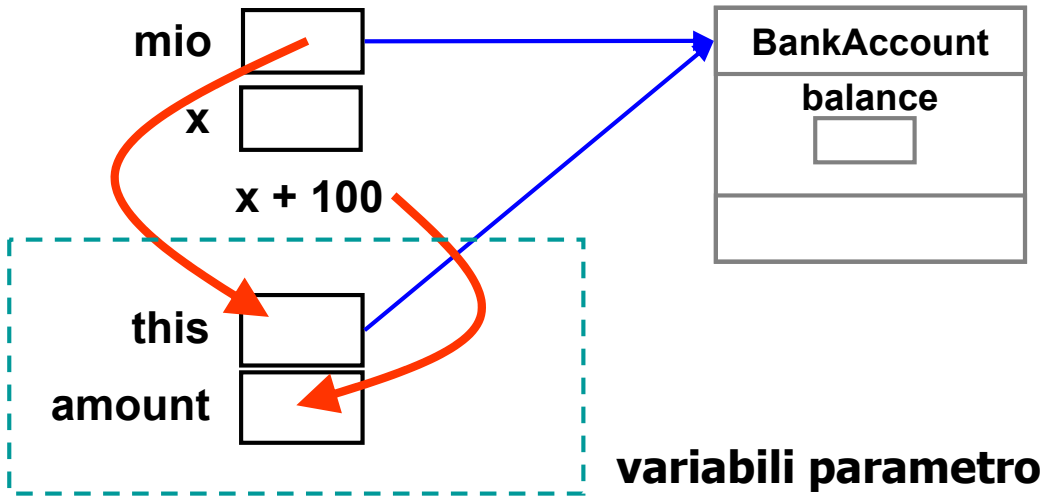
Parola chiave **this**

La parola chiave **this**

- può essere usata solo all'interno del corpo di un metodo;
- produce un riferimento all'oggetto di cui è stato invocato il il metodo;
- costituisce quindi il parametro implicito di metodi e costruttori.

Trasmissione dei parametri per valore

mio.deposit (x+100);



Modifica dei parametri

- Un metodo NON PUÒ modificare un parametro attuale di tipo primitivo
- Un metodo NON PUÒ modificare un parametro di tipo riferimento
- Un merodo PUÒ modificare un oggetto riferito da un parametro

Modifica dei parametri (II)

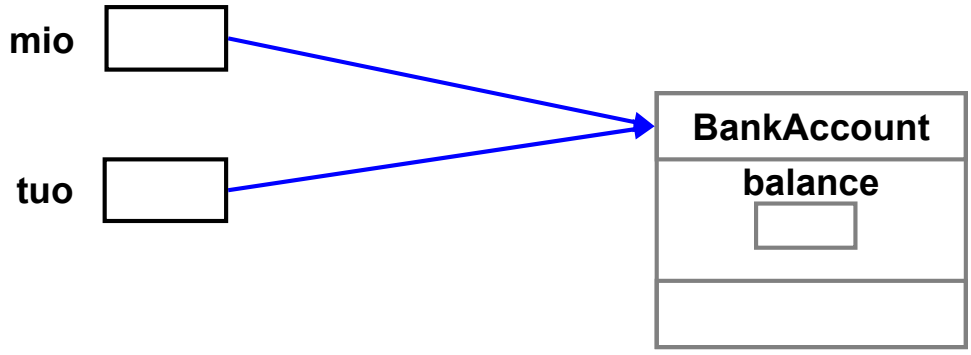
```
public void scambia (Tipo a, Tipo b) {
        Tipo appoggio;
        appoggio = a;
        a = b;
        b = appoggio;
}
...
scambia(x, y); // nessuno scambio
...
```


Modifica dei parametri (III)

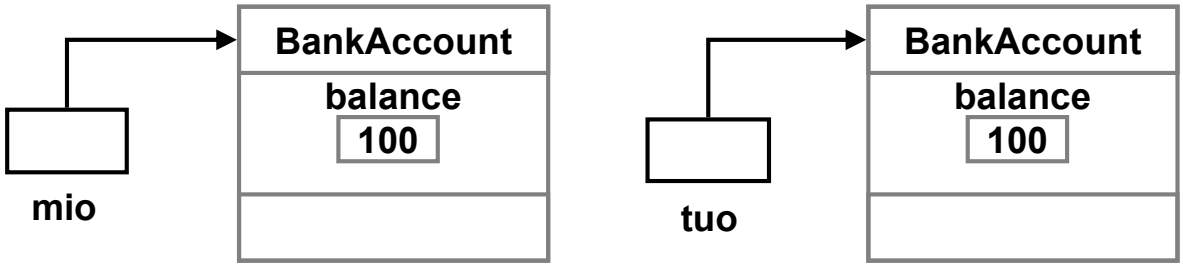
```
class  BankAccount {
   public void  deposit(double amount) {
        balance += amount; }
   public void  withdraw(double amount) {
        balance -= amount; }
   public double  getBalance() {
        return balance; }
   public void transfer(BankAccount other, double amount){
        balance  -= amount;
        other.deposit(amount);  //other.balance += amount;
   }
   private double balance;
}
```

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

Conversioni nelle invocazioni di metodo

- Queste conversioni
  - si verificano al momento della chiamata di un metodo/costruttore e
  - sono applicate al valore di ciascun argomento attuale
- Il tipo dell'argomento viene convertito nel tipo del corrispondente parametro formale
- Le conversioni permesse sono
  - la conversione identità
  - l'estensione di tipo primitivo

static

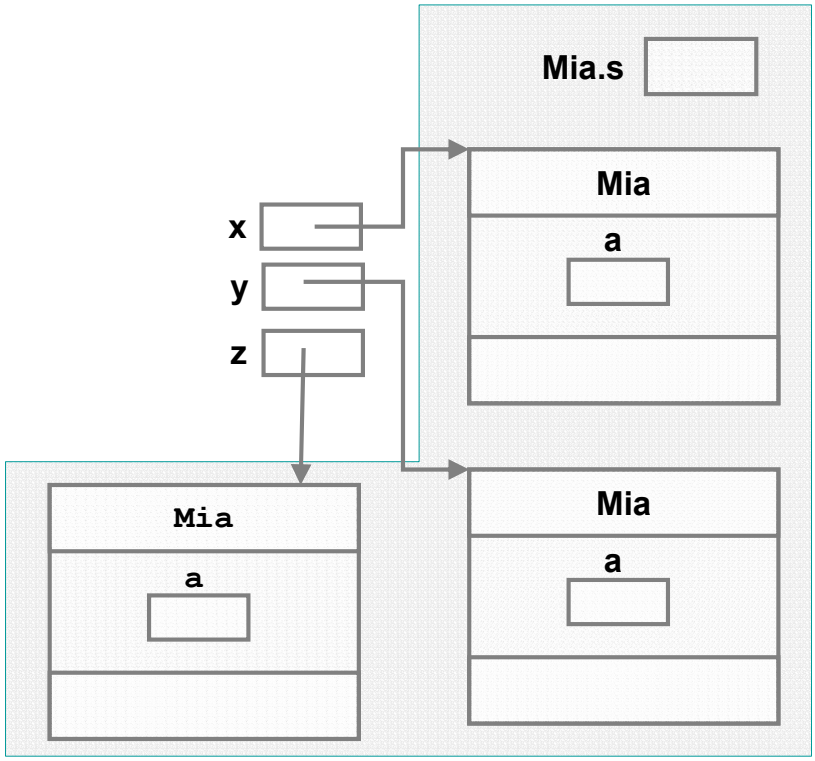
- Un metodo viene detto static, o *metodo di classe*, quando viene dichiarato con il modificatore di accesso **static**
- Un metodo static non ha parametro implicito **this**
  - per cui *non* opererà su una particolare istanza della classe
  - per contro i metodi non-static sono detti *metodi di istanza*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

Campo **static** (I)

- 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

Campo `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 delle variabili istanza

- INIZIALIZZAZIONE AUTOMATICA
- INIZIALIZZAZIONE ESPLICITA
- INIZIALIZZAZIONE TRAMITE IL COSTRUTTORE

Inizializzazione Automatica

Java garantisce un valore iniziale di *default* per ciascun campo:

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

```
classe MiaClasse {
    …
    int i;              // 0
    booelan b;          // false
    char c;             // \u0X0000 (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

```
classe MiaClasse {
    …
    int i = 1;
    int h = f(3);
    int k = g(h);
    // int k = h(b); b non inzializzato
    booelan b = true;
    char c = 'z';
    double d = 2.3;
    AltraClasse a = new AltraClasse();
    …
}
```


Inizializzazione mediante il costruttore

```
classe MiaClasse {
    …
    int i;   // 0
    double d = 2.3;
    public MiaClasse(int a, double b) {
        …
        i = a;
        …
        d = b;
    }
    …
}
```

Inizializzazione delle variabili statiche (I)

- 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

Processo di creazione di un oggetto

L'interprete Java esegue **new T()** attraverso i seguenti passi:

A. se è la prima volta che viene creata un'istanza della classe **T**,
   1. carica la classe **T.class** e
   2. riserva memoria ed inizializza, *una volta per tutte*, le variabili statiche;
B. riserva un'area di memoria per l'oggetto;
C. inizializza quest'area su zero;
D. esegue le inizializzazioni esplicite;
E. esegue il costruttore

Accesso a membri statici

- La prima volta che si accede un membro statico (metodo o variabile) della classe **T**, l'interprete Java
  A. carica la classe **T.class**
  B. riserva memoria ed inizializza, *una volta per tutte*, le variabili statiche
- L'inizializzazione di una variabile statica consiste nell'impostare la variabile al valore di default oppure a quello specificato dall'inizializzatore esplicito

Metodi privati e campi costanti (I)

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

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

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

Metodi privati e campi costanti (II)

```
class BankAccount {
  public BankAccount() {this(0);}
  public BankAccount(double initialBalance) {
      balance = initialBalance; }
  private int static getNewAccountNumber() {
      return ++newAccountNumber;
  }
  // 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 la classe in cui è dichiarata; cessa di esistere quando la classe è 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 diun metodo/costruttore; cessa di esistere quando l'esecuzione 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

Scope (ambito)

- **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
}
```


Top Class

```
class BankAccount { // Top level class
    // membri
}

public class BankAccountDriver {
    public static void main(String args[]) {

        BankAccount mio = new BankAccount();
        mio.deposit(25.8);
    }
}
```

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);
   }
}
```

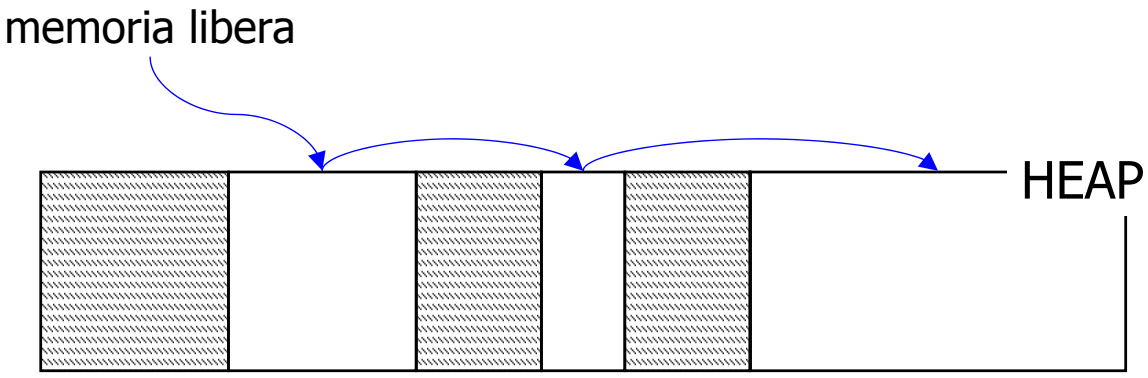
BankAccount può essere testata separatamente da **Applicazione**

Gestione della memoria

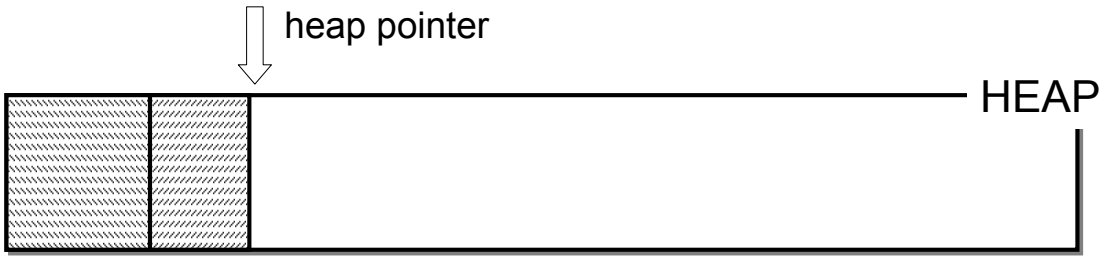
- EFFICIENZA
  - Il numero, l'occupazione di memoria ed il tempo di vita degli oggetti è determinato in fase di compilazione
  - Gli oggetti sono allocati sullo stack o nella memoria statica
  - La distruzione degli oggetti è controllata attraverso le regole di visibilità
  - Gli oggetti sono "poco complicati"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

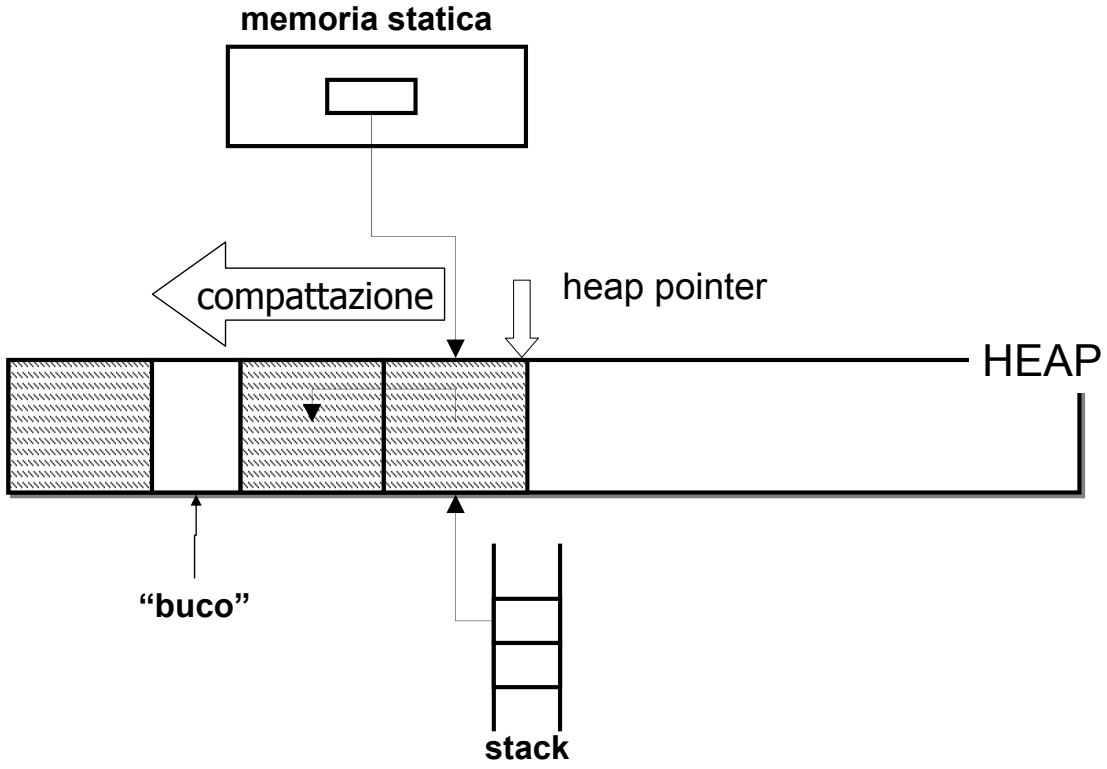


Gestione dello heap



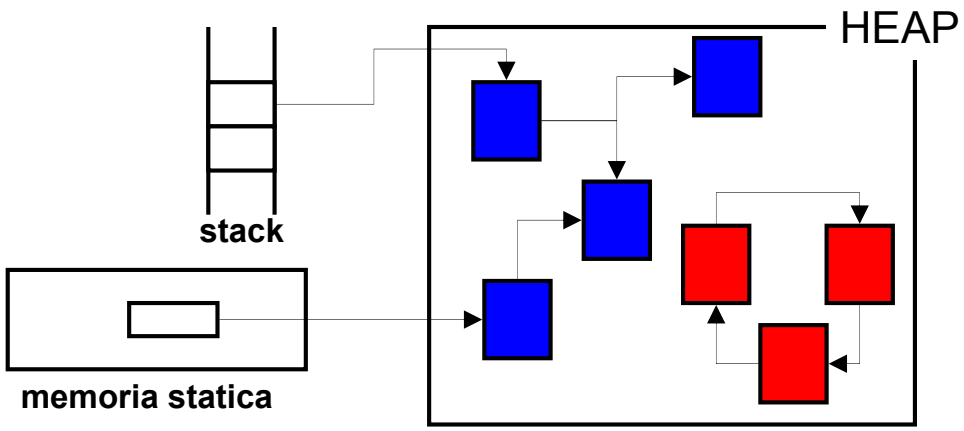
- Allocazione efficiente
- La deallocazione è affidata al supporto run-time del linguaggio: il Garbage Collector
- GC si "accorge" quando un oggetto non è più utilizzato e lo "raccoglie", "reclamando" l'area di memoria riservata all'oggetto
- Il GC gira quando lo ritiene opportuno; mentre gira il GC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

Compattazione



Determinazione oggetti attivi

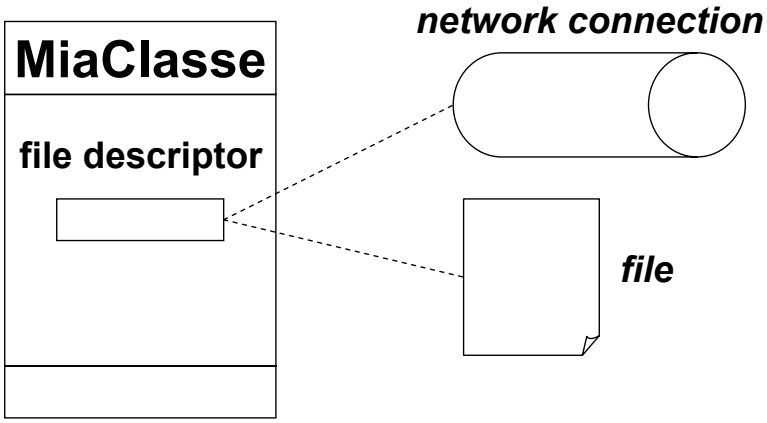
Il GC raccoglie gli oggetti che sono **inattivi**



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 metodo **finalize**

- 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

Conversioni nelle chiamate

- Le chiamate dei metodi costituiscono un contesto di conversione
- Le conversioni permesse sono
  - la conversione identità
  - la conversione per estensione
- Si noti che le costanti di tipo **int** non sono implicitamente ridotte a **byte**, **short** o **char**

Esempio

```
public class Mia {
 public static void f(int a) {
 // corpo di f
 }
 public static void main(String[] args) {
   int i = 1;
   short s = 0X7FFF;
   long n = 2L;
   f(i); // OK: int -> int
   f(s); // OK: short -> int
   f((int)n);
   // f(n); compile-time error
 }
```

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 (I)

- Un metodo è applicabile se e solo se
  - #parametri = #argomenti
  - il tipo di ciascun argomento può essere convertito nel tipo del corrispondente parametro per mezzo di una 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

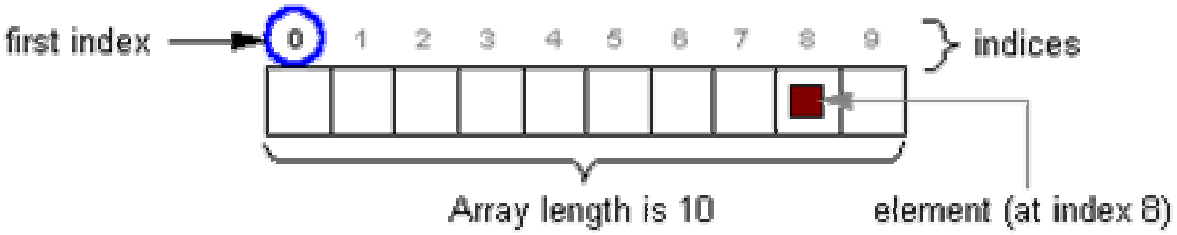
```
class A {
  public static void p() { System.out.print("p() "); p(1); }
  private static void p(int i) { System.out.println("p(int) ");}
}

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
     // q(2.5); compile-time error
     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 errore
- Non è definita l'aritmentica 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≤*expression*≤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 inizializzatore

```
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

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** (**import 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 **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

- 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 "; // stringa vuota
String n = "Roma ";
String i = v + n + 2;
```

- Sottostringhe

```
String s = "Hello "; // stringa vuota
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

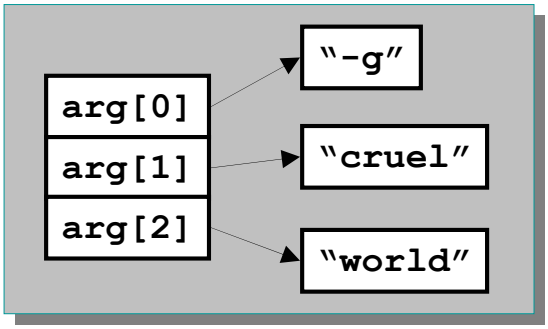
```
String greeting = "Hello";
if ( greeting == "Hello" )
 // probably true
if ( greeting.substring(0, 4) == "Hell" )
 // probably false
```

- Solo le stringhe costanti sono condivise, mentre
- le stringhe ottenute da **+** 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

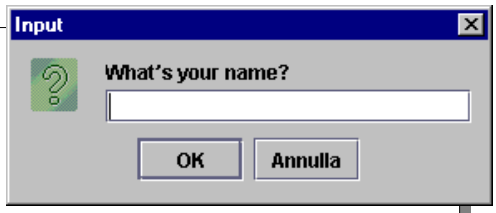
> java ArgsDemo –g cruel world



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

Lettura dell'input




```
import javax.swing.*;

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

     System.exit(0);
  }
}
```