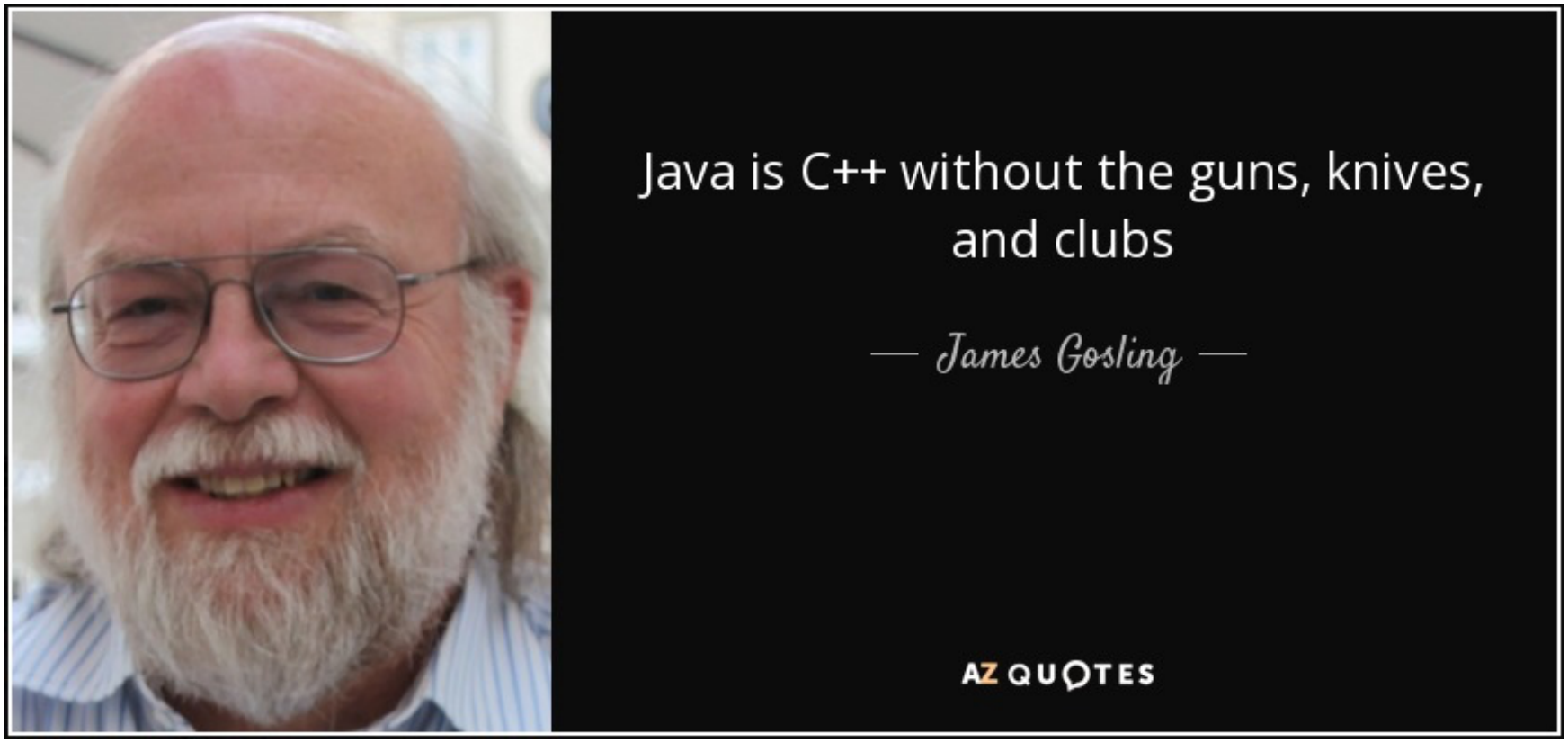
Programmazione ad oggetti **in Java**

- Documentazione JDK e tutorial ufficiale

https://docs.oracle.com/javase

- Libro “Thinking in Java” by Bruce Eckel


Java is C++ without the guns, knives, and clubs
— James Gosling —
AZ QUOTES

Cos'è JAVA?

- Un *linguaggio di programmazione*
  - Impostazione orientata agli oggetti, ispirata originalmente da SmallTalk
  - Sintassi simile al C++
  - Nato nel 1995 diventa popolare con la diffusione del web negli anni 2000
- Un *ambiente di sviluppo*: Oracle JDK + commerciali
- *General-purpose*

I principi [modifica | modifica wikitesto]

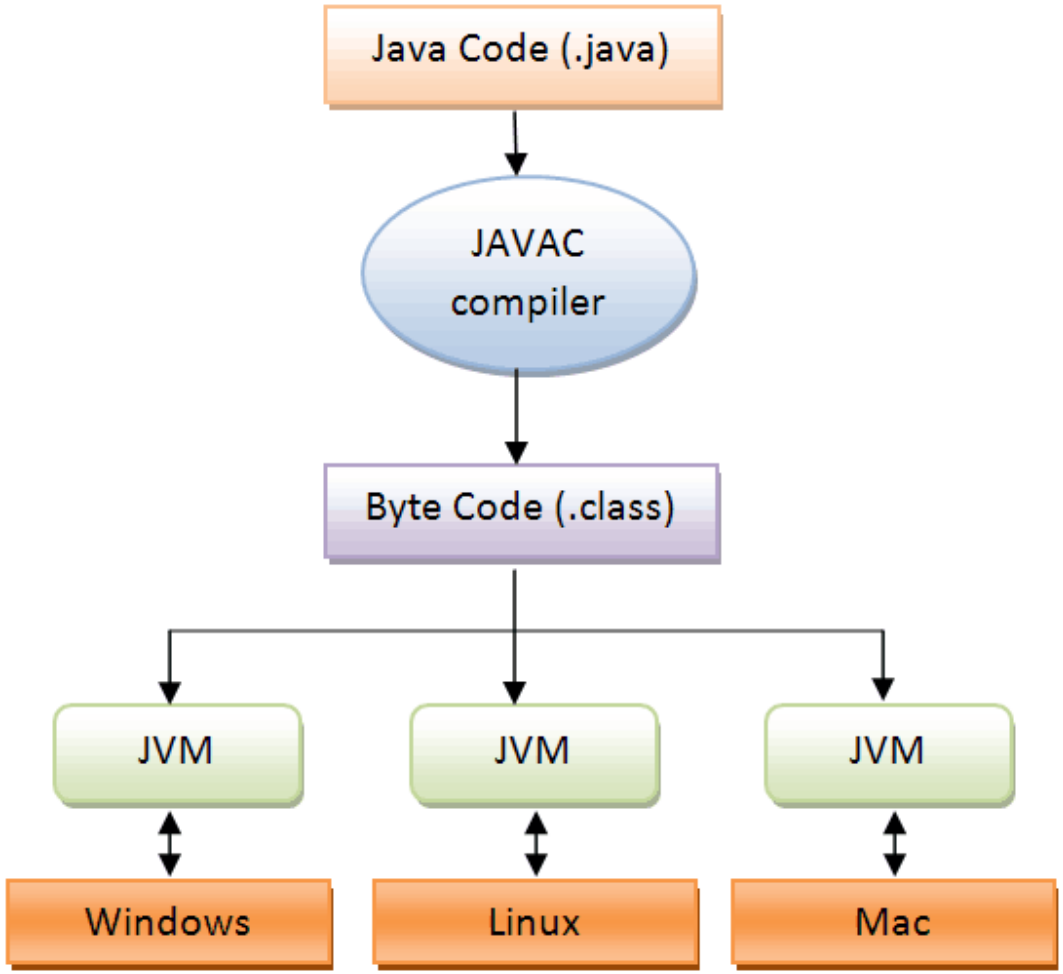
Java venne creato per soddisfare cinque obiettivi primari :[14]

1. essere "semplice, orientato agli oggetti e familiare";
2. essere "robusto e sicuro"
3. essere indipendente dalla piattaforma;
4. contenere strumenti e librerie per il networking;
5. essere progettato per eseguire codice da sorgenti remote in modo sicuro.

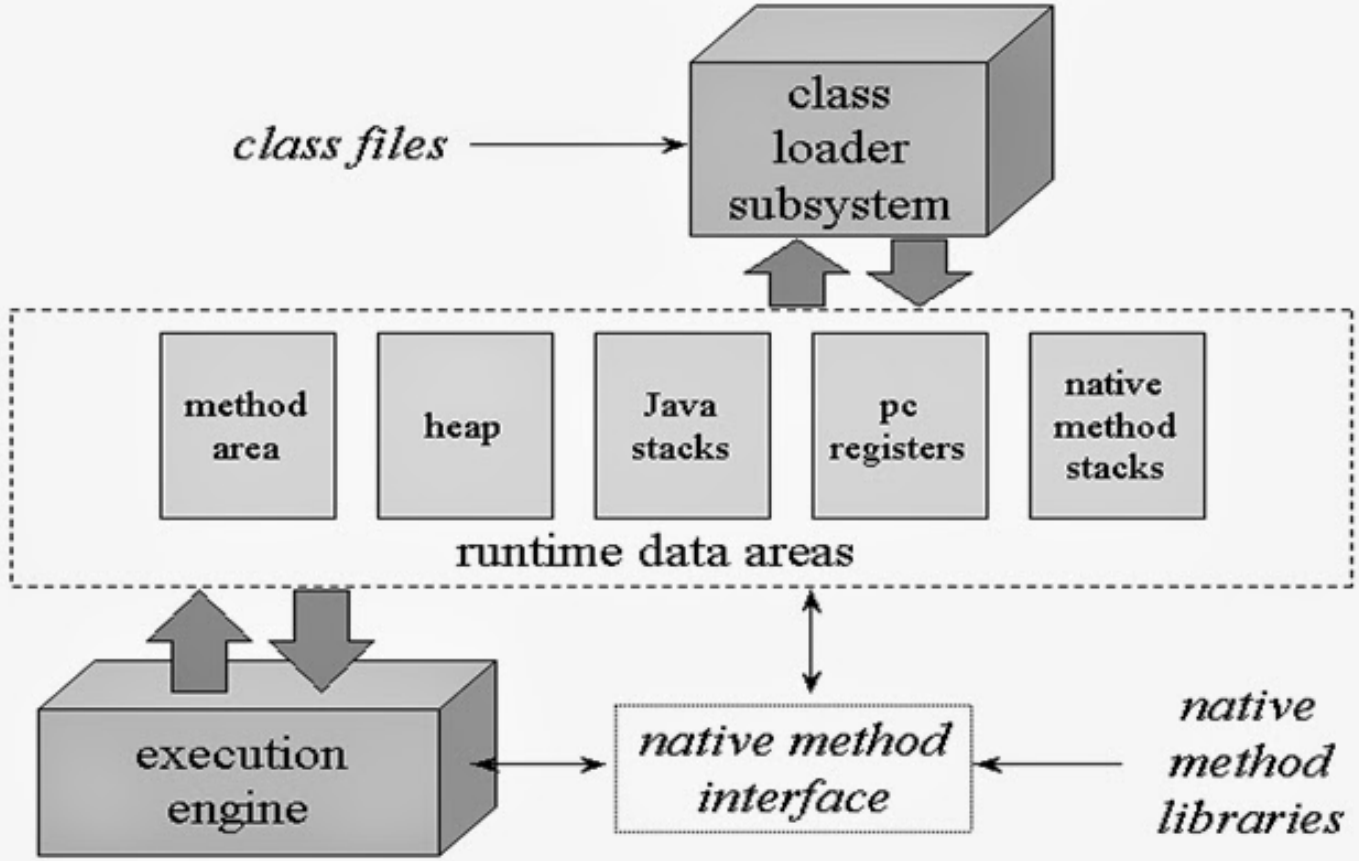
Obiettivi progettuali di Java

- Strumenti usati per raggiungere gli obiettivi
  - Java Virtual Machine (JVM)
  - Automatic Memory Management (garbage collection)
  - Code and type safety

Java Virtual Machine



JVM Architecture Specification



Java Virtual Machine

- Il **bytecode** è il codice della JVM
- La specifica della JVM fornisce definizioni per
  - L'insieme delle istruzioni (l'equivalente di una CPU)
  - L'insieme dei registri
  - Stack
  - Garbage-collected heap
  - Etc.

Bytecode

- Il bytecode mantiene una propria disciplina di tipo
- La maggior parte dei controlli di tipo sono comunque fatti a compile-time sul sorgente
- Ogni interprete Java deve poter eseguire JVM bytecode

Java

```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

Bytecode

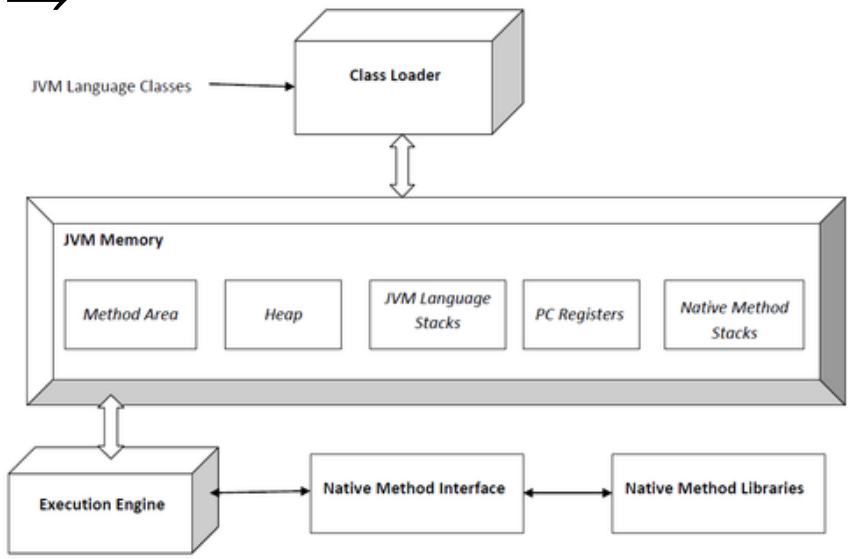
```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge       44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge       31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic       #84; // Field java/lang/System.out:Ljava/io/PrintStream;
34:  iload_1
35:  invokevirtual   #85; // Method java/io/PrintStream.println:(I)V
38:  iinc    1, 1
41:  goto    2
44:  return
```

Memory Management in Java

- Java esenta il programmatore dal compito di deallocare la memoria: c'è la **`new`** ma non la delete
- Un system-level *thread* (un processo concorrente a quello del programma) traccia l'allocazione dinamica della memoria nello heap
- Nei momenti opportuni (ad esempio, idle-cycles della JVM), il garbage collection thread testa e dealloca la memoria che può essere liberata (ovvero gli oggetti privi dei cosiddetti riferimenti)
- La garbage collection avviene quindi **automaticamente a run-time**
- I metodi e l'efficienza della garbage collection possono variare tra diverse implementazioni della JVM

Java runtime environment

- Tre compiti principali
  - Caricamento del bytecode (`.class`) ⇒ Class loader
  - Verifica del codice ⇒ Bytecode verifier
  - Esecuzione del codice ⇒ Runtime interpreter



Class loader [edit]

Main article: Java Class loader

One of the organizational units of JVM byte code is a class. A class loader implementation must be able to recognize and load anything that conforms to the Java class file format. Any implementation is free to recognize other binary forms besides *class* files, but it must recognize *class* files.

The class loader performs three basic activities in this strict order:

1. Loading: finds and imports the binary data for a type
2. Linking: performs verification, preparation, and (optionally) resolution
    - Verification: ensures the correctness of the imported type
    - Preparation: allocates memory for class variables and initializing the memory to default values
    - Resolution: transforms symbolic references from the type into direct references.
3. Initialization: invokes Java code that initializes class variables to their proper starting values.

In general, there are two types of class loader: bootstrap class loader and user defined class loader.

Bytecode verifier

The JVM verifies all bytecode before it is executed. This verification consists primarily of three types of checks:

- Branches are always to valid locations
- Data is always initialized and references are always type-safe
- Access to private or package private data and methods is rigidly controlled

The first two of these checks take place primarily during the verification step that occurs when a class is loaded and made eligible for use. The third is primarily performed dynamically, when data items or methods of a class are first accessed by another class.

Bytecode interpreter and just-in-time compiler [edit]

For each hardware architecture a different Java bytecode interpreter is needed. When a computer has a Java bytecode interpreter, it can run any Java bytecode program, and the same program can be run on any computer that has such an interpreter.

When Java bytecode is executed by an interpreter, the execution will always be slower than the execution of the same program compiled into native machine language. This problem is mitigated by just-in-time (JIT) compilers for executing Java bytecode. A JIT compiler may translate Java bytecode into native machine language while executing the program. The translated parts of the program can then be executed much more quickly than they could be interpreted. This technique gets applied to those parts of a program frequently executed. This way a JIT compiler can significantly speed up the overall execution time.

HelloWorld.java

```
1 //
2 // Programma esempio HelloWorld
3 //
4 public class HelloWorld {
5   public static void main (String args[]) {
6     System.out.println("Hello World!");
7   }
8 }
```

HelloWorld.java

- Se un file `.java` contiene una classe pubblica `C` allora il nome del file deve essere `C.java`
- In un file `.java` può essere definita una sola classe pubblica

HelloWorld.java

```
1 //
2 // Programma esempio HelloWorld
3 //
```

Commenti come in C++

HelloWorld.java

```
4 public class HelloWorld {
```

- Dichiarazione della classe `HelloWorld`

- Un nome di classe `ClassName` specificato

in un sorgente `.java`, una volta compilato con successo, genera un file `ClassName.class` nella stessa directory del sorgente

HelloWorld.java

```
5   public static void main (String args[]) {
```

- L'esecuzione del programma inizia con il metodo `main()` della classe
- Eventuali argomenti del programma sono passati sulla linea di comando al metodo `main()` nell'array `args` di tipo `String`

HelloWorld.java

```
5   public static void main (String args[]) {
```

- `public` ⇒ il metodo `main()` può essere acceduto da chiunque, incluso l'interprete Java
- `static` ⇒ `void main()` è un metodo statico della classe `HelloWorld`.
- `String args[]` ⇒ l'argomento di `main()` è un array di `String`. Ad esempio,

```
% java HelloWorld s1 s2 …
```

HelloWorld.java

```
6       System.out.println("Hello World!");
```

- Stampa la stringa `"Hello World!"` sul dispositivo di standard output (shell): il metodo `println()` è invocato sull'oggetto `System.out` che è un campo dati statico di tipo `PrintStream` della classe `System`

HelloWorld.java

```
7     }
8  }
```

Non serve il ; finale dopo la chiusura } della classe

Compilazione ed esecuzione

- Compilazione
  `javac HelloWorld.java`
- Se il compilatore non ritorna messaggi, `HelloWorld.class` viene creato nella stessa directory
- Esecuzione nella JVM
  `java HelloWorld`

Struttura di un file .java

- Possono apparire, in quest'ordine
  1. Una dichiarazione di **package**
  2. Un qualsiasi numero di dichiarazioni di **import**
  3. Definizioni di **classi** ed **interfacce**

Classes e Packages

- Java include un'ampia API, che implementa le necessità più comuni, non solo per gli usuali compiti di programmazione, ma anche per grafica, networking e molto altro
- Un *package* è un gruppo di classi "omogenee"
- Esempi di package:
  - `java.lang` è il core package. Contiene classi come `String`, `Math`, `Integer`, etc.
  - `javax.swing` per le GUI
  - `java.io`, `java.net`, `java.util` etc.
- Documentazione dell'API è **fondamentale**

Commenti

- Tre stili di commento
  1. `// commento su una linea`
  2. `/* commento su una o più linee */`
  3. `/** commento di` **`documentazione`** `su una o più linee */`
- I **commenti di documentazione** piazzati immediatamente prima di una dichiarazione (di variabile, metodo o classe) indicano che il commento verrà incluso nella documentazione generata automaticamente da `javadoc` per descrivere quella dichiarazione

Java Keywords				
abstract	assert	boolean	break	byte
case	catch	char	class	continue
default	do	double	else	enum
extends	final	finally	float	for
if	implements	import	instanceof	int
interface	long	native	new	package
private	protected	public	return	short
static	strictfp	super	switch	synchronized
this	throw	throws	transient	try
void	volatile	while		
Keywords that are not currently used				
const	goto			

Non c'è un operatore `sizeof`:

dimensione e rappresentazione per tutti i tipi primitivi sono **fissi**

Java Primitive Types

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

8 tipi primitivi

Tipo `boolean`

- Il tipo boolean ha due **valori letterali**: `true` e `false`

```
boolean spia = true;
```

- Il C++ permette di interpretare valori `int` come `boolean`, ovvero esiste una conversione implicita da interi a booleani. Questo **non è permesso** in Java

La classe `String`

- Il tipo `String` **non è primitivo**, è una classe dell'API

```
// dichiara una variabile String e la inizializza
String error = "Out Of Memory \n";
// dichiara due variabili String
String str1, str2;
```

Esempio

```
public class Assegnazioni {
   public static void main (String args[]) {
      int x,y;
      float z = 7.643f;
      double w = 2.9;
      boolean b = true;
      String str = "pippo";
      x = 54; y = 1000;
   }
}
```

Convenzioni usuali per gli identificatori

- `class VariabileComplessa`
- `class ContoBancario`
- `void aggiungiComplesso()`
- `int bilanciaConto()`
- `giuseppeVerdi`
- `MAX_SIZE`

- Definire una classe

```
public class Data {
   int day;
   int month;
   int year;
}
```

- Dichiarare oggetti

```
Data miaData, tuaData;
```

- Accesso ai campi (o membri)

```
miaData.day = 22;
miaData.month = 11;
miaData.year = 1952;
```

Oggetti

- Le **dichiarazioni** di variabili di tipo primitivo provocano l'allocazione della memoria corrispondente
- La sola dichiarazione di una variabile di un tipo classe non provoca l'allocazione dello spazio di memoria per quell'oggetto. Una tale variabile è un cosiddetto ***reference*** (o ***riferimento***) all'oggetto
- In pratica, in molte implementazioni, un reference è un puntatore e la sola dichiarazione di un reference provoca l'allocazione della memoria per memorizzare quel puntatore.
  NOTA BENE: **Java non supporta il tipo puntatore**
- L'allocazione di memoria per gli oggetti avviene **nello heap** tramite una **`new`**
- `Data oggi;` alloca lo spazio per il reference (è quindi un puntatore nullo o non inizializzato)
- `oggi = new Data();` alloca lo spazio effettivo nello heap per l'oggetto, cioè per memorizzare i campi dati della classe `Data`

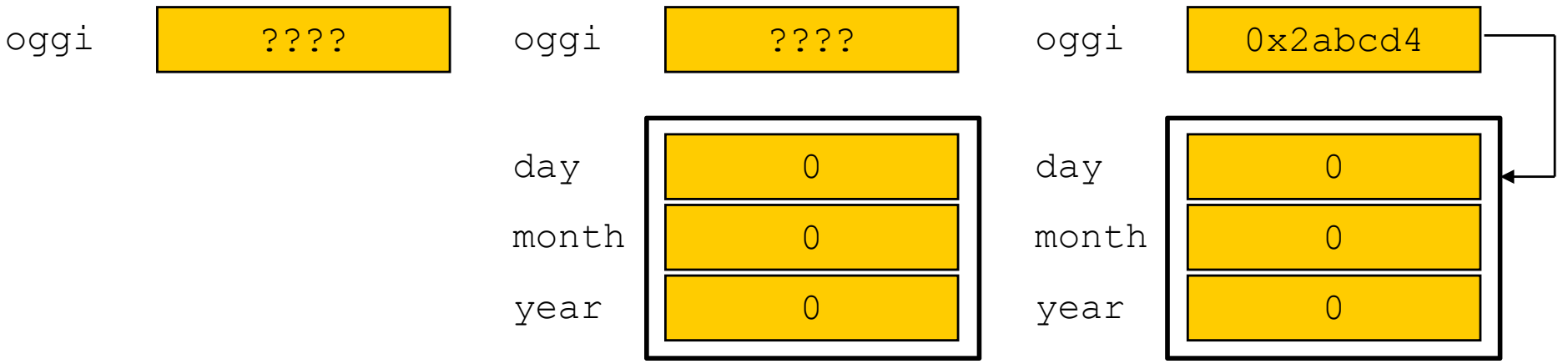
Oggetti

```
Data oggi;
oggi = new Data();
```

```
Data oggi;
oggi = new Data();
```

```
Data oggi;
oggi = new Data();
```

```
Data oggi = new Data();
```

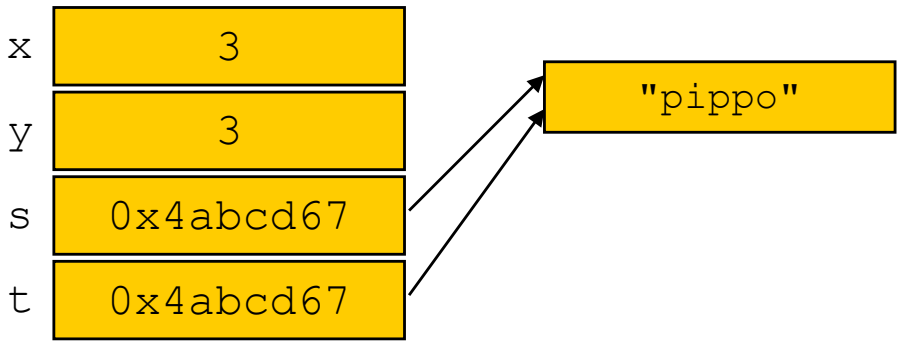


Assegnazioni e references

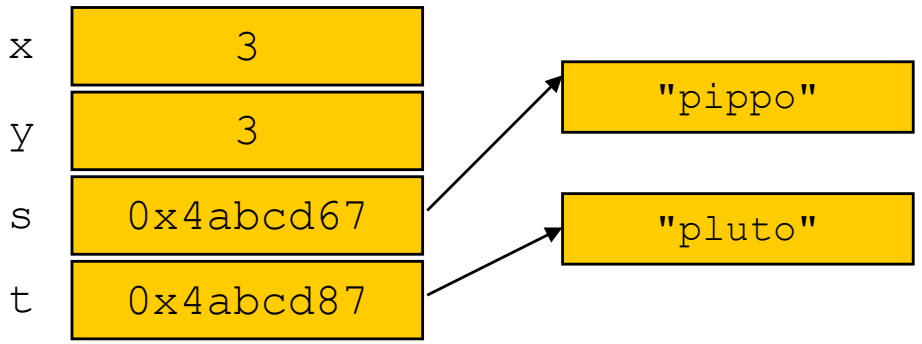
- **Assegnazione ad una variabile di tipo primitivo**
  se `var` è una variabile di qualche tipo primitivo `T` ed `exp` è una espressione di un tipo compatibile con `T` allora l'assegnazione `var=exp;` ha l'usuale comportamento
- **Assegnazione ad una variabile di tipo classe**
  se `r` e `s` sono due riferimenti di qualche classe `C`, allora l'assegnazione `r=s;` provoca la copia del riferimento `s` sul riferimento `r`. Quindi, dopo tale assegnazione, `r` e `s` entrambi si riferiscono allo stesso oggetto, l'oggetto riferito da `s`. L'effetto è quindi identico ad una assegnazione standard tra puntatori o riferimenti in C++

Assegnazioni e references

```
int x = 3;
int y = x;
String s = new String("pippo");
String t = s;
```




```
int x = 3;
int y = x;
String s = new String("pippo");
String t = s;
t = new String("pluto");
```



Valori letterali stringa

- `String s = "pippo";` ha effetti diversi da
  `String s = new String("pippo");`
- `"pippo"` viene detto un *valore letterale stringa.* Ogni letterale stringa è memorizzato in una opportuna area di memoria (nello heap), ed **ogni occorrenza** di un dato letterale stringa può essere pensato come un riferimento a quella specifica area di memoria
- **Quindi:**
- `String s = "pippo";` provoca l'assegnazione al riferimento `s` del riferimento al letterale stringa `"pippo"`
- `String s = new String("pippo");` provoca l'assegnazione al riferimento `s` di una nuova area di memoria contenente la stringa `"pippo"`, che sarà quindi diversa dall'area di memoria del letterale stringa `"pippo"`

Letterali stringa: esempio

```
public class C {
    public static void main(String args[]) {
          String s = "pippo"; String t = "pippo";
          System.out.println(s == t); // true
          String s1 = new String("pippo"); String t1 = new String("pippo");
          System.out.println(s1 == t1); // false
          String p = new String("pippo"); String s2 = p; String t2 = p;
          System.out.println(s2 == t2); // true
          String s3 = new String(p); String t3 = new String(p);
          System.out.println(s3 == t3); // false
    }
}
```

Variabili

- Di tipo primitivo
- Di tipo classe, dette *reference* o *riferimenti* o *oggetti*

Variabili locali

- Variabili definite dentro ad un blocco di un metodo o dentro ad un metodo sono *locali* al blocco o metodo di definizione
- Le variabili locali, di tipo primitivo o reference, sono allocate nello stack quando l'esecuzione raggiunge il blocco, e sono deallocate quando il controllo lascia il blocco
- **Attenzione**: gli oggetti puntati da reference locali sono comunque allocati sullo heap, ed al momento della deallocazione viene deallocato solamente il reference (il puntatore) e non l'oggetto a cui il reference punta (ciò è compito del garbage collector)

Variabili di istanza e di classe

- **Variabili di istanza**: sono i campi dati di una classe C, vengono create quando un oggetto della classe C è costruito tramite una chiamata al costruttore `new C(...)`, ed esistono finchè l'oggetto è referenziato, poi possono venire deallocate dal garbage collector
- **Variabili statiche**: sono dichiarate `static`, vengono create quando viene caricata la classe, e continuano ad esistere finchè esiste la classe, quindi durante tutta l'esecuzione. Le variabili statiche primitive ed i reference statici vengono allocati *nell'area di immagazzinamento statico*, mentre gli oggetti puntati da reference statici vengono comunque allocati sullo heap

Inizializzazione di variabili

- Quando si crea un oggetto di una classe C con un costruttore, i corrispondenti campi dati della classe C sono sempre inizializzati automaticamente al momento dell'allocazione di memoria come in tabella; si tratta della cosiddetta ***inizializzazione automatica "a zero"***
- Un oggetto dichiarato ma non creato è un reference con lo speciale valore **`null`**, ovvero non si riferisce ad alcun oggetto. Se si tenta di usare tale reference si ottiene un'eccezione della classe **`NullPointerException`** (è un tipico e ben noto errore run-time)
- Le variabili locali **devono sempre** essere inizializzate manualmente prima di essere usate in lettura, altrimenti il compilatore segnala un errore

`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0D`
`char`	`'\u0000' (NULL)`
`boolean`	`false`
Tipo reference	`null`

inizializzazione automatica "a zero"

Operatori

- Gli operatori di Java sono sostanzialmente quelli del C++, e vengono usati allo stesso modo
- Ad esempio:

```
[]  .  ++  !  ==  !=  &&  ||  =  +=
```

- `instanceof` è un operatore specifico di Java (operatore di RTTI)

Concatenazione di stringhe

- L'operatore `+` concatena stringhe
- ```
  String prefisso = "Dott. ";
  String nome = "Paolino" + "Paperino";
  String titolo = prefisso + nome;
  ```
- Un argomento deve essere un oggetto o valore di tipo `String`, mentre gli altri argomenti possono anche essere convertiti a `String` (succede che sugli argomenti non di tipo `String` viene invocato automaticamente il metodo `toString()`, quando questo è disponibile)
  ```

Cast numerici

- Quando in un'assegnazione di una variabile con un'espressione, entrambe di tipo primitivo numerico, potrebbe essere persa dell'informazione, il programma **deve sempre** "confermare l'assegnazione" con un *typecast* (conversione di tipo esplicita)
- La sintassi per qualsiasi cast in Java è la sintassi del cast C

```
long longValue = 87L;
int n = (int)(longValue);
int m = 87L; // 87L è long, illegale!
```

- **Attenzione:** i tipi integrali vengono convertiti eliminando i bit più significativi (quelli più a sinistra)

```
short s = 313;     // 16 bit: (313)_10 = (100111001)_2
byte b = (byte)s; // 16 bit => 8 bit: (57)_10 = (111001)_2
System.out.println(s + " " + b); // 313 57
```

Promozioni

- Invece quando in un'assegnazione tra tipi numerici non c'è perdita di informazione (lossless), le variabili **sono sempre automaticamente promosse** (ad esempio, da `int` a `long`)

```
long longValue = 8; // OK, 8 è valore int
float z = 12.356;   // NO, 12.356 è letterale double
double z = 12.356F; // OK, 12.356 è float
```

- Un'espressione exp è compatibile per l'assegnazione ad una variabile var, se il tipo di var è rappresentato con almeno lo stesso numero di bit del tipo di exp

Promozioni

- Quindi, le conversioni implicite tra i tipi primitivi sono **tutte e sole** le seguenti:

`byte` $\Rightarrow$ **`short`** $\Rightarrow$ **`int`** $\Rightarrow$ **`long`** $\Rightarrow$ **`float`** $\Rightarrow$ **`double`**

Istruzioni condizionali

- ```
  if (boolean expression) {
      blocco;
  }
  ```
- ```
  if (boolean expression) {
      blocco;
  } else {
      blocco;
  }
  ```
- Differenza con il C++: `if()` richiede un'espressione di tipo **`boolean`**, non numerica. Inoltre, tipi Booleani e numerici non possono essere convertiti implicitamente: bisogna quindi usare **`if (x != 0)`**

Dichiarazione di array

- Si possono dichiarare array di qualsiasi tipo, primitivo o classe:

```
char s[];
Point p[]; // Point è una classe
char[] s;  // sintassi alternativa equivalente
Point[] p; // da preferirsi (tipo array Point[])
```

- In Java, **un array è un oggetto**. Quindi, come per tutte gli oggetti, la dichiarazione non crea l'array, ma crea solamente un reference

Creazione di array

- **`char s[]; s = new char[20];`**
  prima dichiara e poi crea un array di 20 `char`
- **`int[] ia = new int[3];`**
  dichiara e crea un array di 3 `int`
- **`p = new Point[200];`**
  crea un array di 200 `Point`. **Attenzione:** **Non crea** 200 oggetti `Point`, ma crea solamente 200 reference ad oggetti della classe `Point`! Questi oggetti devono essere costruiti esplicitamente:
  `p[0] = new Point(); p[1] = new Point();`…
- Non viene fissata la lunghezza dell'**oggetto** array: un nuovo array di lunghezza differente potrebbe essere assegnato a `s` o `p`
- Come in C++, l'indice parte da zero. Ogni volta che viene utilizzato un indice viene verificato a **run-time** che esso appartenga all'intervallo appropriato
- Non esiste il concetto di array statico del C++: la lunghezza dell'array può sempre non essere nota a compile-time

Inizializzazione di array

- Quando si crea un array con una new, ogni suo elemento viene opportunamente inizializzato a zero (come per i campi dati di una classe)
- **`s = new char[20];`**
  crea un array di 20 `char`, ed ogni `s[i]` è inizializzato al carattere zero (`` `\u0000` ``, carattere nullo)
- **`p = new Point[200];`**
  crea un array di 200 variabili di tipo `Point`, ed ogni `p[i]` è inizializzata a `null`, ovvero non si riferisce ancora ad un effettivo oggetto di tipo `Point`. Ciò potrà essere fatto successivamente mediante `p[i] = new Point();`

Inizializzazione di array

Java permette la creazione ed inizializzazione simultanea di array, analogamente al C++

```
String[] nomi = {
        "pippo",
        "pluto",
        "paperino"
};
```

è equivalente a:

```
String[] nomi = new String[3];
nomi[0] = "pippo";
nomi[1] = "pluto";
nomi[2] = "paperino";
```

Altro esempio:

```
Point[] p = {
        new Point(),
        new Point(),
        new Point()
};
```

Array multidimensionali

- Poichè gli array possono essere di qualsiasi tipo, è possibile definire array di array (e array di array di array, etc.)

```
int[][] mat = new int[3][];
mat[0] = new int[5];
mat[1] = new int[5];
mat[2] = new int[5];
```

- La prima istruzione definisce un array di tre elementi. Ogni elemento è un reference `null` ad un elemento di tipo `int[]`, ed ognuno deve essere inizializzato
- **Nota:** `int[][] mat = new int[][4]` non ha quindi senso ed è illegale!

Array multidimensionali

Conseguenza: è possibile definire array *non rettangolari:*

```
int[][] mat_nr = new int[3][];
mat_nr[0] = new int[2];
mat_nr[1] = new int[4];
mat_nr[2] = new int[6];
```

Gli array rettangolari sono i più comuni. Per loro si può usare la seguente sintassi

```
int[][] mat = new int[3][5];
```

```
double[][] dm = {
        { 1.0, 0.0, 0.0 },
        { 0.0, 1.0, 0.0 },
        { 0.0, 0.0, 1.0 },
};
```

Array bounds

Gli indici degli array iniziano da 0. Il numero di elementi di un array è memorizzato nel campo dati **`length`** dell'oggetto array. Viene eseguito il "run-time bound checking" per testare gli accessi "out-of-bounds":

```
int[] list = new int[58];
for (int i = 0; i < list.length; i++) {
      list[i] = i + 10;
}
```

Array e reference

Una volta costruito un array, e fissata quindi la sua lunghezza, questa **non può più essere cambiata**. Comunque, una variabile array come `int[] list` non è altro che un reference, e quindi è possibile farla puntare ad un altro array:

```
int[] list = new int[58];
list = new int[258];
```

In questo esempio, il primo array viene "perso", cioè diventa garbage, a meno che non esista un qualche altro riferimento verso di lui

Copie di array

Java mette a disposizione nella classe `System` dell'API il metodo statico `arraycopy()` per copiare array:

static void **arraycopy**(**Object** src, int srcPos, **Object** dest, int destPos, int length)

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

```
int[] a1 = {1,2,3,4,5,6};
int[] a2 = {10,9,8,7,6,5,4,3,2,1};
System.arraycopy(a1,0,a2,0,a1.length);
// a2 = {1,2,3,4,5,6,4,3,2,1}
```

Per array di oggetti, `arraycopy()` copierà reference, non oggetti. Gli oggetti stessi, naturalmente, non cambiano

Classi e oggetti

- Naturalmente Java supporta le tre caratteristiche fondamentali di un linguaggio ad oggetti:
  - *Encapsulation* (incapsulamento)
  - *Polymorphism* (polimorfismo)
  - *Inheritance* (ereditarietà)
- Anche in Java le classi permettono di realizzare il concetto generale di *abstract data type (ADT)*

Passaggio per valore

- In Java **tutti** i parametri dei metodi vengono passati **per valore**. Quindi, i valori dei parametri formali in un metodo sono copie dei valori dei parametri attuali specificati all'invocazione. Quindi in una invocazione i parametri attuali non possono in alcun modo essere modificati dal metodo
- **Nota Bene:** L'affermazione precedente è vera perchè quando il parametro formale è un riferimento ad un oggetto, si passa "per valore" il riferimento e non l'oggetto stesso. Quindi il metodo può modificare l'oggetto a cui il parametro si riferisce, ma non può invece modificare il riferimento stesso
- Strettamente parlando, in Java non esiste il passaggio dei parametri per riferimento
- D'altra parte, visto che ogni variabile di tipo classe è sempre un riferimento, ciò significa che ogni oggetto è sempre passato per riferimento!

```
public class Prova {
        float f;
        public void changeInt (int n) { n = 55; }
        public void changeStr (String s) {
          s = new String ("pippo");
        }
        public void changeObj (Prova ref) { ref.f = 99.0F; }

        public static void main (String args[]) {
                String str = new String("pluto"); int val = 11;
                Prova pt = new Prova();
                pt.changeInt(val);
                System.out.println(val);  // stampa: 11
                pt.changeStr(str);
                System.out.println(str);  // stampa: pluto
                pt.f = 13.0F;
                pt.changeObj(pt);
                System.out.println(pt.f); // stampa: 99.0
        }
}
```

`this` è un reference all'oggetto di invocazione di un metodo che si può usare nel corpo di metodi non statici

```
public class Data {
        int day, month, year;
        public void tomorrow() {
                this.day = this.day + 1;
                // ...
        }
}
```

Nel frammento sopra, come al solito l'uso di `this` è ridondante e normalmente si usa: `day = day +1;`

A volte l'uso di `this` è necessario, tipicamente per passare l'oggetto di invocazione come un argomento di un metodo:

```
public class Data {
        int day, month, year;
        public void born() {
                DataDiNascita d = new DataDiNascita(this);
                d.mandaAnnunci();
        }
}
```

Per ogni campo dati e per ogni metodo di una classe Java è necessario specificare il corrispondente modificatore di accesso; la mancanza di una parola chiave tra {`public`, `protected`, `private`} corrisponde ad un quarto specifico modificatore di accesso che analizzeremo nel seguito. Come al solito, il modificatore `private` premesso ad un membro di una classe C rende quel membro *inaccessibile* da qualsiasi codice eccetto ai metodi della stessa classe C

In Java non esistono i file header di intestazione. Una libreria viene fornita con la documentazione (che corrisponde in qualche modo ai file header del C++) ed il bytecode delle classi. La documentazione di una classe descrive quindi i membri non privati di una classe.

Overloading di metodi

Come in C++, Java permette l'*overloading* dei nomi di metodi di una classe

```
public void println(int i)
public void println(float f)
public void println()
```

Due regole (come in C++) per i metodi sovraccaricati

1) La lista degli argomenti attuali in una chiamata di un metodo sovraccaricato deve permettere la **determinazione non ambigua** del metodo da chiamare, altrimenti il compilatore segnala l'ambiguità della chiamata
2) Il diverso tipo di ritorno di due metodi sovraccaricati non può essere la sola differenza (ovvero, le liste dei parametri formali devono essere diverse)

Esempio di ambiguità:

```
public void m(float f, double d) {};
public void m(double d, long l) {};
object.m(1.23F,45L); // NON COMPILA!
```

`new` e costruttori

- **`new C(...)`**: costruisce nello heap un nuovo oggetto della classe `C`
- Una **`new C(...)`** provoca i seguenti effetti:
  1) Allocazione della memoria, e corrispondente inizializzazione automatica (a **`null`** o "a 0")
  2) Viene eseguita ogni eventuale *inizializzazione esplicita*
  3) Viene invocato il *costruttore* (esplicito o standard)

Inizializzazioni esplicite

Le dichiarazioni dei campi dati di una classe possono avere delle *inizializzazioni esplicite* che vengono effettuate prima della invocazione di un costruttore:

```
public class Inizializzata {
        private int x = 65;
        private String name = "Gastone";
        private Data d = new Data();
        ...
}
```

Costruttori

L'inizializzazione esplicita è un meccanismo semplice per definire dei valori iniziali per i campi dati di un nuovo oggetto. Il meccanismo dei metodi *costruttori* è più duttile e standard

```
public class C {
        // campi dati
        public C() { // costruttore senza argomenti
                // costruzione dell'oggetto ...
        }
        public C(int i) { // costruttore ad 1 argomento intero
                // costruzione dell'oggetto ...
        }
}
```

Delegation nei costruttori

Quando si hanno più costruttori, questi possono anche chiamarsi in modo "annidato", uno dentro l'altro. Ciò si ottiene usando `this(...)` come chiamata per un metodo costruttore

```
public class Dipendente {
          private String nome;
          private int stipendio;

          public Dipendente(String n, int s) {
            nome = n; stipendio = s;
          }

          public Dipendente(String n) {
            this(n,0);
          }

          public Dipendente() {
            this("Unknown");
          }
}
```

Regola importante: In ogni definizione di costruttore, l'eventuale chiamata `this` ad un altro costruttore deve necessariamente essere il primo statement

Attenzione: in Java **non** esiste la lista di inizializzazione per un costruttore

Il costruttore di default standard

- Come in C++, se in una classe non viene definito alcun costruttore, è comunque disponibile il ***costruttore di default standard***
- Il costruttore di default standard è senza argomenti ed ha il corpo vuoto. Pertanto, viene chiamato con `new C()`, ed il suo comportamento semplicemente consiste nell'allocare lo spazio per l'oggetto, nel provocare l'inizializzazione automatica ("a zero") dell'oggetto e nell'eseguire le eventuali inizializzazioni esplicite
- Quando si definisce esplicitamente almeno un costruttore, il costruttore di default standard non è più disponibile. Quindi, una chiamata `new C()` provocherà un errore in compilazione se non è definito esplicitamente un costruttore senza argomenti

Valori di default

- **Attenzione:** Java **non** prevede i valori di default per i metodi
- In particolare, non è possibile seguire la prassi C++ di definire dei costruttori con valori di default

Ereditarietà

L'ereditarietà in Java si ottiene tramite la parola chiave `extends`

```
public class Dipendente {
        String nome;
        Data dataDiAssunzione;
        Data dataDiNascita;
        ...
}
public class Dirigente extends Dipendente {
        String titolo;
        Dipendente[] subordinati;
        ...
}
```

Al solito, si usa anche la notazione $\leq$ di sottotipo: `Dirigente` $\leq$ `Dipendente`

Nota: Come al solito, nella documentazione la descrizione di un campo dati o metodo ereditato esiste solo nella classe di definizione di quel membro. Nella documentazione di una sottoclasse appare la lista dei campi dati e dei metodi ereditati da qualche supertipo

Ereditarietà singola

- A differenza del C++, Java permette solamente *l'ereditarietà singola* (*single inheritance*): una classe può estendere **una sola** altra classe
- Nella comunità OO, ci sono discussioni sui relativi meriti dell'ereditarietà multipla (una classe può estendere più classi simultaneamente) e singola. La motivazione Java è: "*single inheritance makes code more reliable*"
- Le *interfacce* sono uno strumento Java per sopperire alla mancanza dell'ereditarietà multipla

Costruttori nelle sottoclassi

- Una sottoclasse eredita tutti i campi dati e tutti i metodi dalla sua superclasse (classe padre)
- Come in C++, **non "eredita"** gli eventuali costruttori
- Una sottoclasse può quindi avere un costruttore in due soli modi:
  1. Viene definito un costruttore esplicito
  2. Non viene definito alcun costruttore, e quindi è disponibile il costruttore standard di default
- Vedremo che un costruttore di una sottoclasse tipicamente invoca il costruttore della superclasse