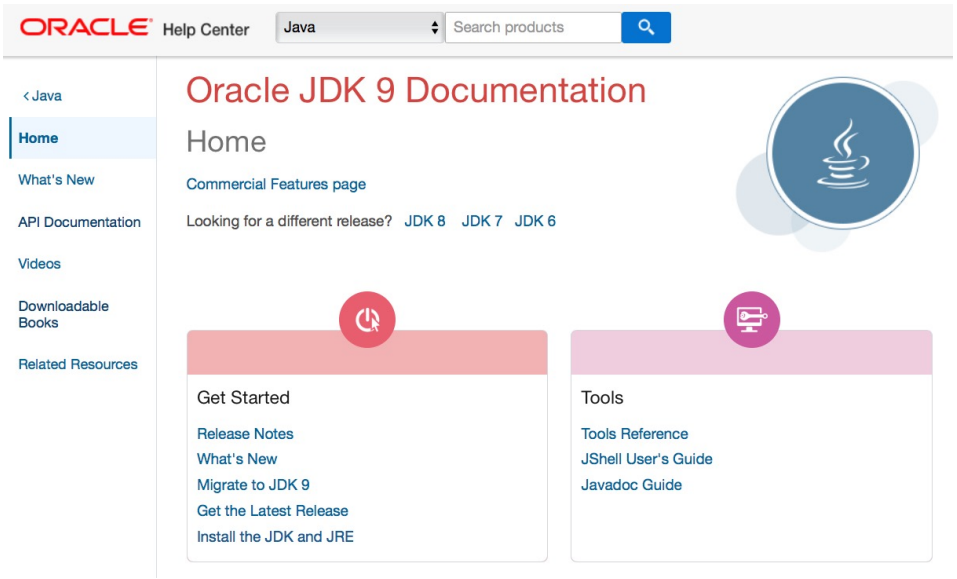
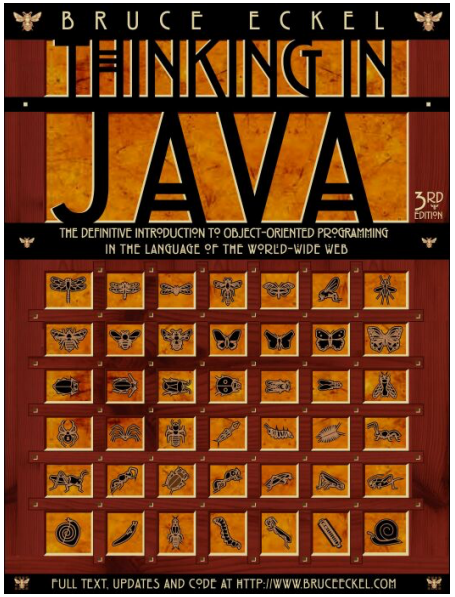
Programmazione ad oggetti **in Java**

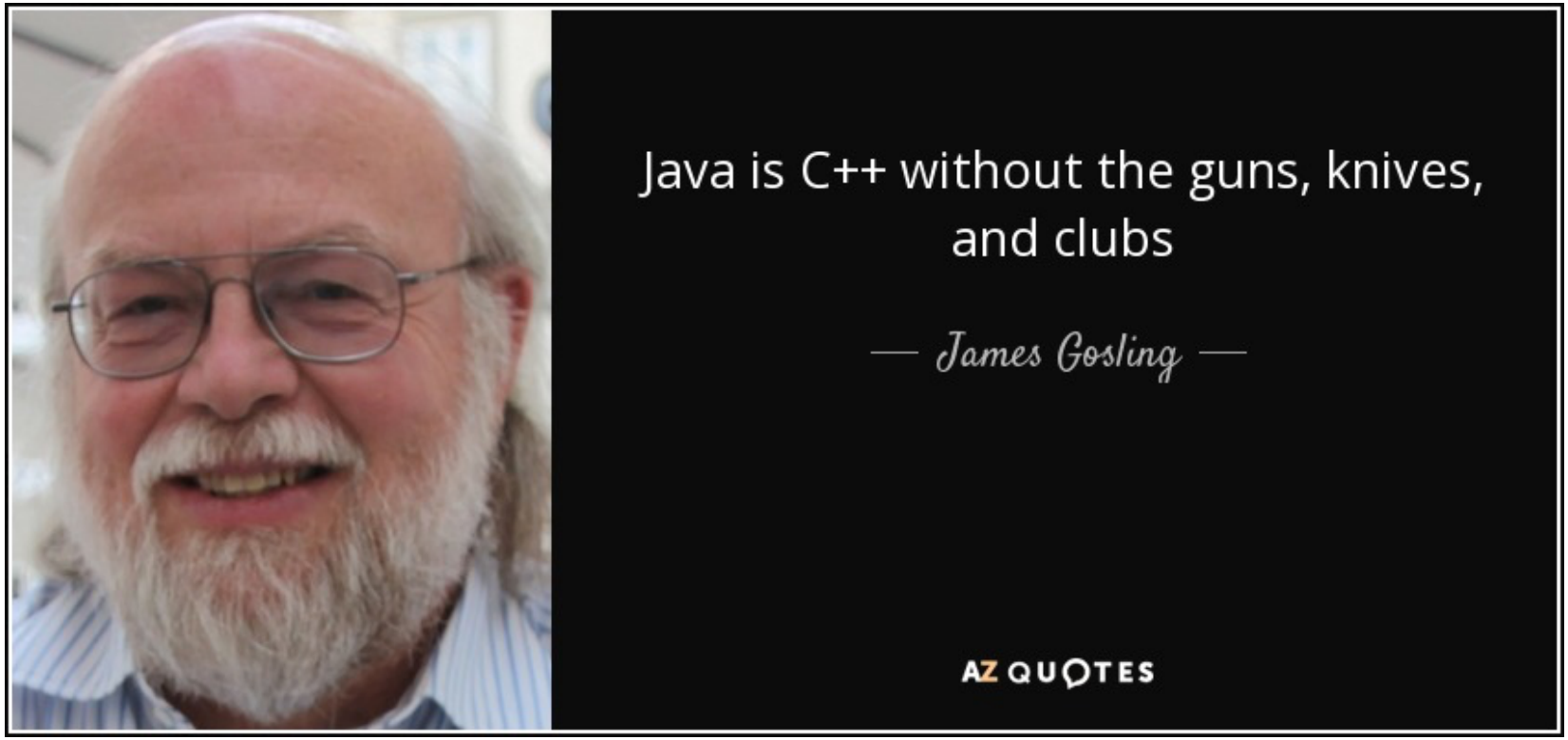
- Documentazione JDK e tutorial ufficiale

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]

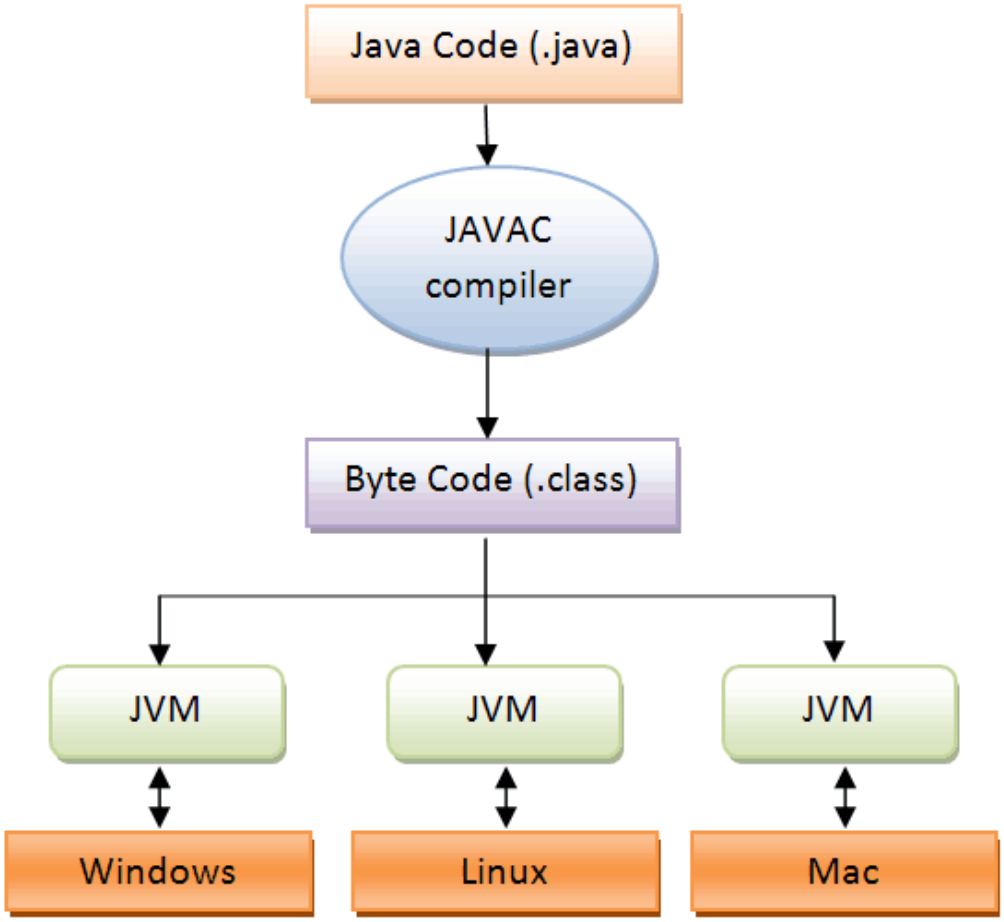
Java venne creato per soddisfare cinque obiettivi primari :[14]

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.

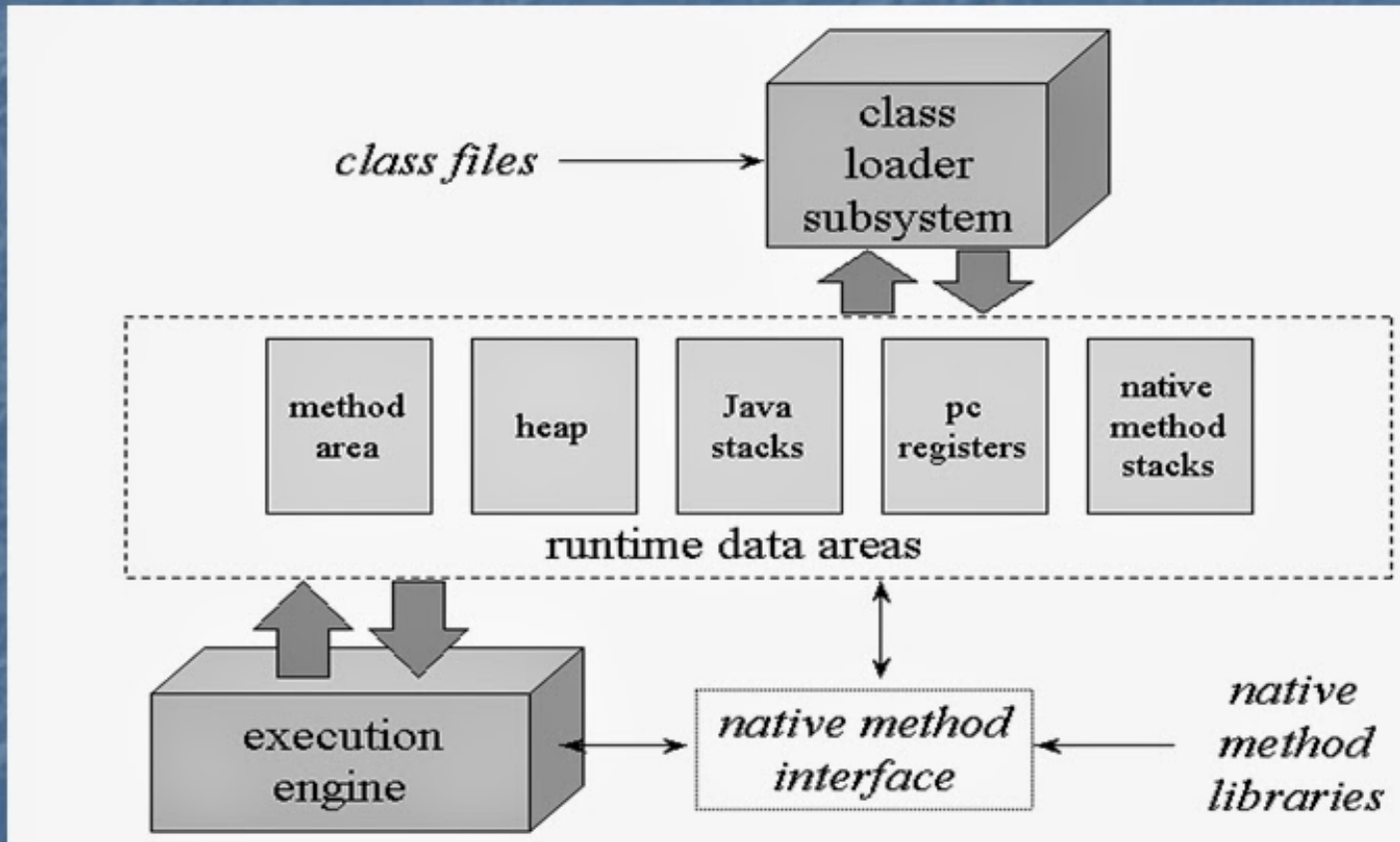
Obiettivi progettuali di Java

- 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

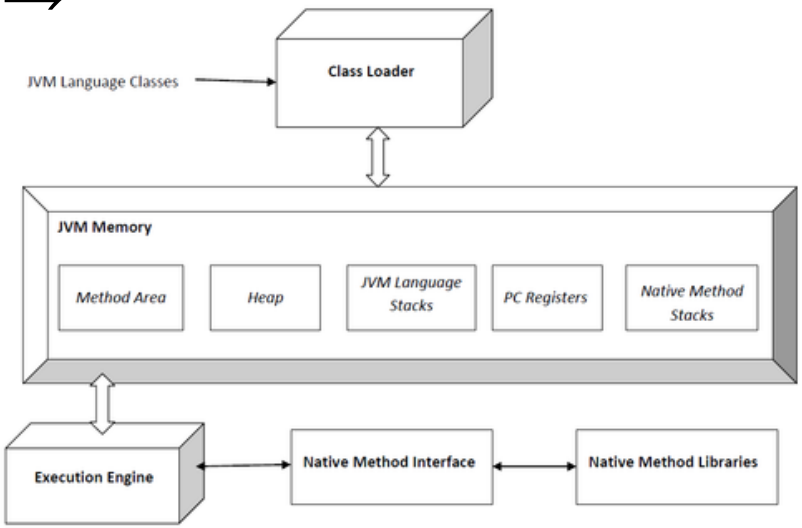
```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge       44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge       31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic       #84; // Field 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`) $\Rightarrow$ Class loader
  - Verifica del codice $\Rightarrow$ Bytecode verifier
  - Esecuzione del codice $\Rightarrow$ 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`

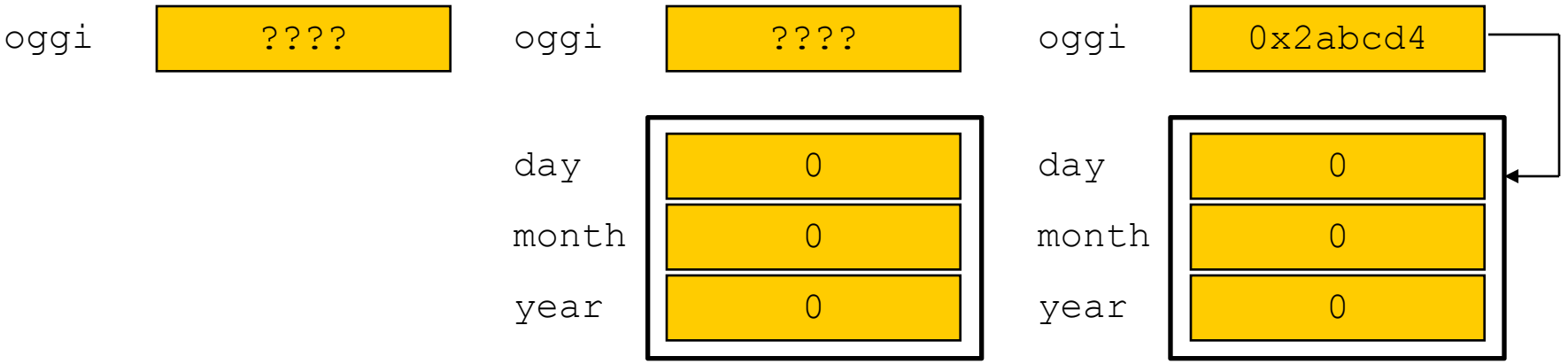
Oggetti

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

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

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

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

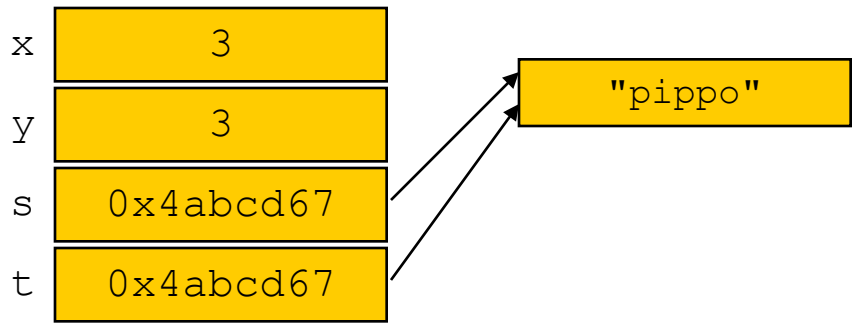


Assegnazioni e references

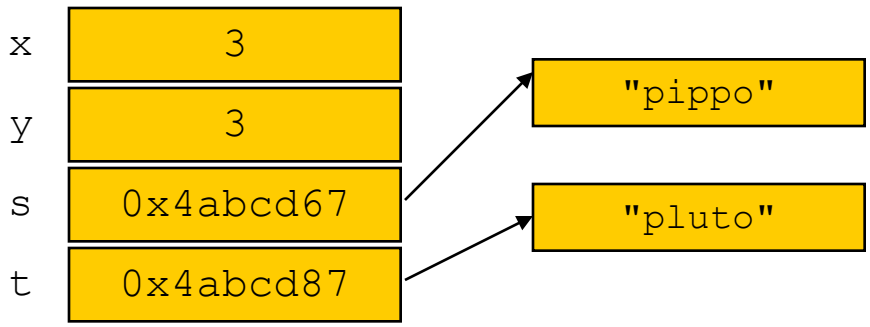
- **Assegnazione ad una variabile di tipo primitivo**
  se `var` è una variabile di qualche tipo primitivo `T` ed `exp` è una espressione di un tipo compatibile con `T` allora l'assegnazione `var=exp;` ha l'usuale comportamento
- **Assegnazione ad una variabile di tipo classe**
  se `r` e `s` sono due 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 costruisce un array con una new, ogni suo elemento **viene 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`** e "a 0"
  2) Viene eseguita ogni eventuale ***inizializzazione esplicita***
  3) Viene **invocato il *costruttore*** (sia esso 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

Una classe derivata in Java si definisce mediante la keyword **`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"** 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

Polimorfismo

Un oggetto vero e proprio è **monomorfo**, cioè ha una sola "forma", il tipo al momento della sua costruzione. Una variabile di qualche tipo classe, cioè un reference, invece è **polimorfa** poichè può riferirsi a oggetti di tipo diverso

Subtyping rule

In qualsiasi contesto dove sia richiesto una espressione di tipo T è possibile usare una espressione che abbia un tipo S che è *sottotipo* di T, cioè tale che $S \leq T$.

Come al solito, scriveremo $S < T$ quando vale:

$S \leq T$ e $S \neq T$.

Subtyping

Per i **tipi primitivi** abbiamo già visto la definizione della relazione di subtyping $\leq$. Inoltre, la relazione di subtyping è *riflessiva*, cioè per ogni tipo primitivo T vale $T \leq T$

Subtyping

Per i tipi classe (cioè non primitivi) valgono le seguenti regole di definizione della relazione di subtyping $\leq$.

Siano T1 e T2 due tipi classe. Allora T1 $\leq$ T2 se vale una delle seguenti condizioni:

1) T1 e T2 sono lo stesso tipo (riflessività)
2) T1 è definito come "class T1 extends T2"
3) T1 è un tipo array T1a[], T2 è un tipo array T2a[] e vale T1a $\leq$ T2a
4) Esiste un tipo T3 tale che T1 $\leq$ T3 e T3 $\leq$ T2 (transitività)
5) T2 è il tipo `Object`

Aggiorneremo nel seguito la relazione di subtyping con nuove regole

Object

- **Novità rispetto a C++:** in Java esiste la classe **`java.lang.Object`** che è superclasse di ogni classe (quindi anche di ogni tipo array!). La classe **`java.lang.Object`** è la classe radice di ogni gerarchia di classi. Quindi per ogni classe C, vale C ≤ **`Object`**
- Una definizione di classe come

  `public class Dipendente`

  potrebbe essere pensata come una abbreviazione di

  `public class Dipendente extends Object`
- La classe `Object` definisce parecchi metodi utili (si veda la documentazione, si possono segnalare `clone()`, `equals()`, `getClass()`, `notify()`, `wait()`, `...`), tra cui il metodo `toString()`

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
protected Object	**clone**() Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	**getClass**() Returns the runtime class of this Object.
int	**hashCode**() Returns a hash code value for the object.
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
String	**toString**() Returns a string representation of the object.
void	**wait**() Causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object.
void	**wait**(long timeout) Causes the current thread to wait until either another thread invokes the **notify()** method or the **notifyAll()** method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos) Causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

`toString()`

`public String toString()` ritorna una rappresentazione stringa dell'oggetto di invocazione. Ciò spiega perchè in Java "tutto" può essere convertito ad una rappresentazione `String` (sebbene l'implementazione di `toString()` in `Object` ritorni una stringa che potrebbe essere di utilità limitata)

toString

```
public String toString()
```

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Upcasting

Upcasting: è indotto dalla relazione di subtyping tra classi. In qualsiasi contesto dove è richiesto un oggetto di qualche classe C, è sempre possibile usare un oggetto di una sottoclasse di C. Come in C++, l'azione di convertire un riferimento di una classe in un riferimento di una sua qualsiasi superclasse viene chiamata upcasting. L'upcasting è sempre una conversione sicura

```
Dipendente dip = new Dirigente();
```

Attenzione: tramite la variabile `dip` è possibile accedere **solamente** ai membri dell'oggetto `Dirigente` che sono parte di `Dipendente`, mentre i membri specifici di `Dirigente` sono inaccessibili

```
dip.titoloDirigenziale = "Capo del personale"; // Illegale!
```

Argomenti polimorfi

In particolare, per upcasting, un metodo con un parametro formale di una certo tipo **`T`**, potrà sempre essere invocato con un parametro attuale di un qualsiasi sottotipo di **`T`**

```
public TaxRate findTaxRate(Dipendente dip) {
       // calcola il tasso fiscale di dip
}

// in qualche metodo:
Dirigente d = new Dirigente();
TaxRate t = findTaxRate(d);
```

Collezioni eterogenee

Collezioni *omogenee* sono composte di oggetti tutti dello stesso tipo. Grazie alla classe **`Object`** ed al polimorfismo, è possibile definire ***collezioni eterogenee*** di oggetti di classi diverse. Ad esempio, un array di **`Object`** può contenere oggetti di qualsiasi tipo, naturalmente non variabili di tipo primitivo (vedremo che le Wrapper class in qualche modo permettono anche ciò):

```
Dipendente[] staff = new Dipendente[128];
staff[0] = new Dirigente();
staff[1] = new Dipendente();
...
```

Tipo statico e tipo dinamico

- Il tipo statico di un reference `ref` è il tipo di dichiarazione di `ref`
- Per subtyping un reference `ref` può riferirsi ad oggetti che hanno un tipo diverso dal tipo statico di `ref`
- Il tipo dinamico di un reference `ref` è il tipo dell'oggetto a cui effettivamente `ref` si riferisce in un dato istante dell'esecuzione
- Per un qualsiasi reference `ref` useremo la seguente notazione:

  TS(`ref`)=tipo statico di ref

  TD(`ref`)=tipo dinamico di ref
- Il tipo statico di un reference non può cambiare a run-time, mentre, naturalmente, può cambiare il suo tipo dinamico
- Vale sempre la relazione: TD(ref) $\leq$ TS(ref)

Tipo statico e tipo dinamico

```
class C {}
class D extends C {}
class E extends D {}

public class Client {
   public static void main(String[] args) {
      C r1 = new C(); // TS(r1)=C, TD(r1)=C
      C r2 = new E(); // TS(r2)=C, TD(r2)=E
      D r3 = new D(); // TS(r3)=D, TD(r3)=D
      E r4 = new E(); // TS(r4)=E, TD(r4)=E
      r2 = r3;        // TD(r2)=D
      r1 = r2;        // TD(r1)=D
      r3 = r4;        // TD(r3)=E
      r2 = new E();   // TD(r2)=E
   }
}
```

RTTI

- L'identificazione dei tipi a tempo di esecuzione (RTTI) consente di determinare il tipo dinamico di un reference. RTTI è piuttosto sofisticata in Java e riveste un ruolo importante
- Vi sono 3 forme di RTTI
  1) Il downcast tra reference, che **è sempre dinamico** (cioè eseguito a run-time) ed utilizza RTTI per accertarsi che il cast sia corretto e genera una eccezione se si tenta di effettuare un cast non corretto
  2) L'utilizzo della classe (dell'API) **`Class<T>`** e dei suoi metodi (incluso `isInstance(Object)`)
  3) L'operatore **`instanceof`**

L'operatore **`instanceof`**

Data un qualsiasi reference **`ref`** ed una qualsiasi classe **`C`**, la sintassi dell'operatore **`instanceof`** è la seguente:

`ref instanceof C`

e ritorna un valore di tipo `boolean`

Comportamento:

`ref instanceof C` ritorna **`true`** se e soltanto se

1) **`ref`** è un reference non nullo e
2) TD(**`ref`**) ≤ **`C`**

L'operatore `instanceof` permette quindi di determinare se il tipo dinamico di un qualsiasi riferimento `ref` è compatibile con un qualsiasi tipo `C`

L'operatore `instanceof`

```
public class Dipendente {...}
public class Dirigente extends Dipendente {...}
public class Agente extends Dipendente {...}

public void someMethod(Dipendente dip) {
    if (dip instanceof Dirigente) {
      // ad esempio, ottiene certi benefici
    }
    if (dip instanceof Agente) {
      // ad esempio, ottiene certi rimborsi
    }
    ...
}
```

Downcast di oggetti

Sia ref un riferimento per cui vale TD(ref) < TS(ref). Quindi siamo in una situazione in cui il tipo dinamico di ref è un sottotipo stretto del suo tipo statico. In tal caso, tramite un *downcast*, è possibile definire un riferimento ref2 di un tipo statico C che "raffina" il tipo statico di ref e che si riferisce allo stesso oggetto a cui ref si riferisce: per il tipo C dovrà quindi essere vero che

$$TD(ref) \leq C \leq TS(ref)$$

Supponiamo quindi di sapere che valga $TD(ref) \leq C \leq TS(ref)$. In tal caso, possiamo effettuare un safe downcast nel seguente modo:

```
C ref2 = (C)ref;
```

Downcast di oggetti

Perchè fare un downcast?

La risposta è la stessa del C++: con un (safe, cioè sicuro) downcast di un reference ref riusciamo ad ottenere le ulteriori funzionalità offerte dalla classe C sottotipo di TS(ref)

Downcast di oggetti

Naturalmente, prima di effettuare un downcast del reference ref ad un tipo target C è fondamentale essere sicuri che ciò sia safe (cioè corretto), ovvero che valga la relazione TD(ref) $\leq$ C

Possiamo ottenere questa informazione tramite RTTI, tipicamente con l'operatore **`instanceof`**: infatti l'espressione (ref instanceof C) ritorna true se e soltanto se TD(ref) $\leq$ C.

```
public void someMethod(Dipendente dip) {
    if (dip instanceof Dirigente) {
        Dirigente d = (Dirigente)dip;
        System.out.println("Dirige l'ufficio " +
          d.ufficioDiretto()); // possibile solo dopo il cast
    }
    ...
}
```

Downcast di oggetti

Senza il test RTTI di **`instanceof`** si corre il rischio di un fallimento a run-time del cast: verrebbe sollevata una eccezione di tipo **`ClassCastException`**

Class ClassCastException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.ClassCastException

All Implemented Interfaces:

Serializable

```
public class ClassCastException
extends RuntimeException
```

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a `ClassCastException`:

```
Object x = new Integer(0);
System.out.println((String)x);
```

Cast di oggetti

Il cast di riferimenti obbedisce alle seguenti regole:

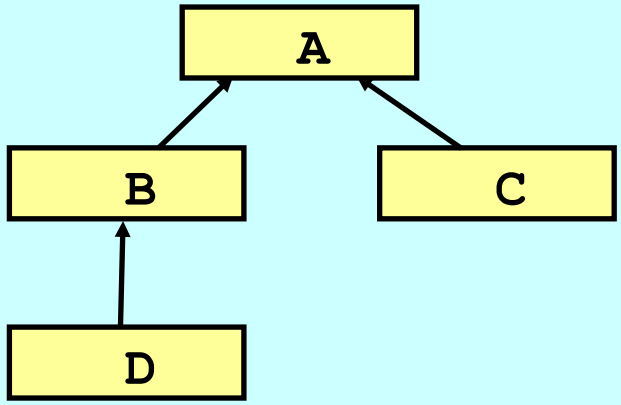
1) L'**upcasting**, cioè da sottotipo a supertipo in una gerarchia, è sempre permesso, ovvero è un cast implicito. Infatti non richiede l'operatore di cast, e si può sempre effettuare tramite una assegnazione. Più precisamente: l'assegnazione "s=r;" compila se e soltanto se TS(r) ≤ TS(s)
2) Il compilatore **considera legale qualsiasi downcasting da supertipo a sottotipo** nella gerarchia. Più precisamente: se C ≤ TS(r) allora il cast "(C)r" compila correttamente. Ad esempio, un cast da `Dirigente` ad `Agente` è legale staticamente.
3) Una volta che il compilatore accetta un downcasting, il **cast viene comunque sempre effettuato a run-time**: se il cast non ha successo viene sollevata una eccezione di tipo **`ClassCastException`**
4) Vedremo nel seguito le regole del casting per riferimenti che abbiano come tipo statico una interfaccia

Esempio

```
class A { }
class B extends A { }
class C extends A { }
class D extends B { }

public class E {
    public static void main(String[] Args) {
        A a = new A(); B b = new B();
        C c = new C(); D d = new D();
        A r = new B();
        A s = new D();
        a = b; // OK, upcast
        a = d; // OK, upcast
        /* b = c; */ // ILLEGALE
        b = (B)r; // OK, downcast
        d = (D)r; // downcast: compila, ma lancia un'eccezione
        d = (D)s; // OK, downcast
        /* b = s; */ // ILLEGALE
        /* d = (D)c; */ // ILLEGALE
        b = (B)s; // OK, downcast
        b = (D)s; // OK, downcast

    }
}
```



Overriding di metodi

Come previsto dal paradigma ad oggetti, una sottoclasse può modificare il comportamento predefinito di qualche sua superclasse ridefinendone qualche metodo ereditato.

Se un metodo definito in una classe C ha gli stessi **nome**, **lista dei tipi degli argomenti** e **tipo di ritorno** (cioè la stessa **segnatura**) di un metodo di una superclasse B di C, allora si dice che questo nuovo metodo in C ***ridefinisce*** (***overrides***) il corrispondente metodo in B.

Esempio

```
public class Dipendente {
    protected String nome;
    int stipendio;
    public String getDetails() {
        return "Nome: " + nome + "\n" + "Stipendio: " + stipendio;
    }
}

public class Dirigente extends Dipendente {
    String ufficioDiretto;
    public String getDetails() {
        return "Nome: " + nome + "\n" + "Dirige: " + ufficioDiretto;
    }
}
```

Overriding di metodi

Differenza importante con il C++: Supponiamo che una classe base B includa più definizioni sovraccaricate di un metodo m(). Allora la ridefinizione del nome di metodo m() in una classe derivata da B **non nasconde** tutte le versioni di m() in B, ma solamente quella con la stessa segnatura.

Quindi, l'overloading di un nome di metodo m() in una classe C funziona senza considerare se il metodo m() è definito in C oppure in una classe base di C

Esempio

```
class B {
  char m(char c) {
    System.out.println("char m(char)");
    return 'd';
  }
  float m(float x) {
    System.out.println("float m(float)");
    return 1.0f;
  }
}

class C {}

class D extends B {
  void m(C r) { System.out.println("void m(C)"); }
}

public class H {
  public static void main(String[] args) {
    D d = new D();
    d.m(1);          // stampa: float m(float)
    d.m('x');        // stampa: char m(char)
    d.m(2.0f);       // stampa: float m(float)
    d.m(new C());    // stampa: void m(C)
  }
}
```

Come vengono invocati i metodi in Java?

Invocazione statica o dinamica?

```
Dipendente dip = new Dipendente();
Dirigente dir = new Dirigente();
dip.getDetails(); // invoca getDetails() di Dipendente
dir.getDetails(); // invoca getDetails() di Dirigente


Dipendente dip = new Dirigente();
dip.getDetails(); // cosa invoca?
```

Invocazione dinamica

- **Regola di invocazione:** In ogni invocazione di metodo ref.m() viene sempre invocato il metodo m() della classe TD(ref).
- **L'invocazione dei metodi in Java è sempre dinamica**: viene invocato il metodo appartenente alla classe corrispondente al tipo dinamico del reference di invocazione
- In Java quindi il binding tra riferimento di invocazione e metodo da invocare non è mai statico ma avviene sempre a run-time: il meccanismo di invocazione è **sempre il late binding** (o dynamic binding)
- Quindi in Java **tutti i metodi** (non statici) **sono implicitamente virtuali**
- Questa scelta progettuale segue il **principio della semplicità**: è disponibile un'unica metodologia di invocazione e quindi necessariamente sarà l'invocazione virtuale. Svantaggio noto: il late binding comporta un overhead a run-time

Overriding

Regole per l'overring dei metodi

1) Se ridefinisco un metodo m() mantenendo la stessa segnatura (lista ordinata dei tipi degli argomenti) allora il **tipo di ritorno** deve essere **esattamente lo stesso** del metodo originale, altrimenti il compilatore segnale una illegalità
2) Un metodo ridefinito **non può diventare meno accessibile** del metodo originale (ad esempio `public` non può diventare `private` o `protected`)
3) Un metodo ridefinito **non può sollevare tipi di eccezioni diversi** da quelli del metodo originale

Esempio

```
class B {
    public int m(int x) {return 2;}
}

public class C extends B {
    public double m(int x) {return 3.0;}
    public static void main(String args[]) {
       B b = new B(); C c = new C(); B r = new C();
       System.out.println(b.m(1) +" "+ c.m(1) +" "+ r.m(1) );
    }
}
/* NON COMPILA:
m(int) in C cannot override m(int) in B; attempting to use
incompatible return type
found: double   required: int
*/
```

Esempio

```
class A {
    public int m(int x) {return 2;}
}
class B extends A {
    private int m(int x) {return 3;}
}
public class C {
    public static void main(String args[]) {
       A ref = new B(); System.out.println(ref.m(1));
       // verrebbe invocato dinamicamente un metodo privato!
    }
}
/* NON COMPILA:
m(int) in B cannot override m(int) in A; attempting to assign
weaker access privileges; was public
    private int m(int x) {return 3;}
*/
```

`super`

Nei metodi di istanza di qualche classe D derivata direttamente da B, una chiamata **`super.m()`** **invoca sempre il metodo m() della superclasse B**, escludendo l'eventuale ridefinizione del metodo **`m()`** in D oppure in qualche sottoclasse di D. Vi è quindi **static binding** per **`super.m()`**

Il riferimento **`super`** può essere utilizzato solamente per invocare metodi. Naturalmente, in una chiamata **`super.m()`**, non è necessario che **`m()`** sia definito nella superclasse diretta B: esso potrebbe anche essere stato ereditato da qualche altra classe che si trovi più in alto di B nella gerarchia

Esempio

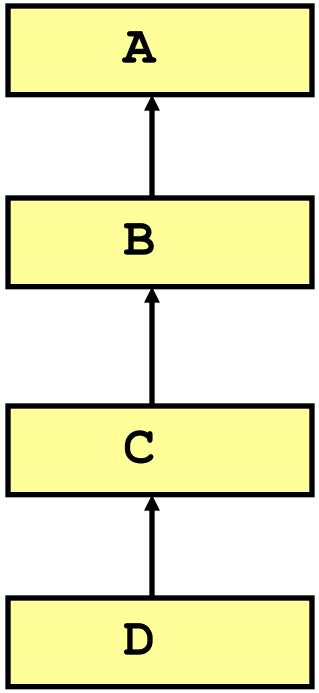
```
public class Dipendente {
    private String nome;
    private int stipendio;

    public String getDetails() {
        return "Nome: " + nome + "\nStipendio: " + stipendio;
    }
}

public class Dirigente extends Dipendente {
    private String ufficioDiretto;

    public String getDetails() {
        return super.getDetails() + "\nDirige: " + ufficioDiretto;
    }
}
```

Esempio




```
class A {
    public int m() {return 1;}
    public char n() {return 'A';}
}
class B extends A {
    public char n() {return 'B';}
}
class C extends B {
    public int m() {return 3;}
}
public class D extends C {
    public int m() {return super.m();}
    public char n() {return super.n();}
    public static void main(String args[]) {
        A ref = new D();
        System.out.println(ref.m() + " " + ref.n());
        ref = new C();
        System.out.println(ref.m() + " " + ref.n());
        ref = new B();
        System.out.println(ref.m() + " " + ref.n());
    }
} // STAMPA:
// 3 B
// 3 B
// 1 B
```

Costruttori nelle sottoclassi

Supponiamo che D sia una classe derivata direttamente da B. Allora quando viene costruito un oggetto di D, la rappresentazione di questo oggetto "contiene" un **sottooggetto** della classe base B. Quindi, come in C++, i costruttori nelle classi derivate devono in qualche modo richiamare i costruttori delle classi base.

Quando viene costruito un oggetto della classe derivata D con una **`new`**:

1) Viene chiamato un **costruttore**, esplicitamente o implicitamente, per la **classe base diretta di D** (se questa esiste)
2) Viene allocata la memoria per i campi dati propri di D e viene inizializzata automaticamente (a **`null`** o "a 0")
3) Vengono eseguite le eventuali inizializzazioni esplicite di D
4) Viene eseguito il corpo del costruttore di D

Costruttori nelle sottoclassi

Possiamo pensare che ogni classe ha sempre una classe base, dal momento che eventualmente questa può essere Object. L'invocazione **esplicita** di un costruttore della classe base avviene mediante una chiamata

```
super(arg1, arg2, ...)
```

Se si definisce un costruttore che non ha chiamate esplicite `super(...)` oppure chiamate ad un altro costruttore della classe D tramite un'invocazione `this(...)`, allora il compilatore inserisce **automaticamente una chiamata implicita al costruttore di default** (quindi `super()`) ella classe base diretta di D. Se tale costruttore non è disponibile, il compilatore segnala un errore.

Quando sono usati in un costruttore, `super` o `this` devono necessariamente essere la **prima istruzione**. Quindi, `super` e `this` sono **mutuamente esclusivi**, ovvero non possono essere presenti entrambi. La motivazione è chiara: una chiamata this(...) delega il compito di invocare un costruttore della classe base ad un altro costruttore e quindi non può essere seguita da una chiamata super(...), mentre una chiamata super(...) invoca esplicitamente un costruttore della classe base e quindi non può essere seguita da una chiamata this(...) che potrebbe pure invocare un costruttore della classe base.

L'ordine delle inizializzazioni esplicite dei campi dati di una classe è quello testuale di dichiarazione dei campi dati nella classe.

Esempio

```
public class Dipendente {
    private String nome;

    public Dipendente(String n) {
        nome = n;
    }
}

public class Dirigente extends Dipendente {
    private String ufficioDiretto;

    public Dirigente(String n, String u) {
        super(n); // chiamata del costruttore padre con parametro n
        ufficioDiretto = u;
    }
}
```

Esempio

```
class C { // manca il costruttore di default
    public C(int x) {}
    public C(int a, int b) {}
}

class D extends C {
    public D() {super(1,2);}
    // public D(int a) {} // chiamata implicita super() => ILLEGALE
    public D(int a) {this();} // OK
}

public class E extends D {
    public E() {this(2);}
    public E(int x) {super(x);}
    // public E(int a, int b) {super(a); this(b);} // ILLEGALE
}
```

Esempio

```
class Z {
    int z;
    Z(int i) {z=i; System.out.print(" Z:" + i);}
}
class C {
    Z i=new Z(2);
    Z j=new Z(3);
    C() {i=new Z(4);}
}
public class D extends C {
    Z x=new Z(5);
    Z y=new Z(6);
    D() {super(); x=new Z(7);}
    public static void main(String args[]) {
        D d = new D();
        System.out.println("\n" + d.i.z +" "+ d.j.z
                           +" "+ d.x.z +" "+ d.y.z);
    }
}
// STAMPA:
// Z:2 Z:3 Z:4 Z:5 Z:6 Z:7
// 4 3 7 6
```

Package

Il meccanismo Java dei **`package`** fornisce un modo per "raggruppare" delle classi. Utilità dei package:

- Permettono di raggruppare classi (e interfacce) con **funzionalità logicamente correlate**
- Nei package si possono definire classi e membri di classe visibili (cioè disponibili) solo all'interno del package stesso. Ovvero esiste una **modalità di accesso a *"livello di package"***

Package

Se il file contenente il codice di una classe (o più classi) include una **dichiarazione di `package`**, cioè una dichiarazione di appartenenza della classe ad un certo package, questa deve apparire all'inizio del file (sono ammessi commenti che precedono la dichiarazione). È permessa **una sola dichiarazione di package** per file sorgente, e questa governa l'intero sorgente **`.java`**

I nomi di **package sono gerarchici** e separati da punti, convenzionalmente in minuscolo. Esempi dall'API: `java.util`, `java.util.concurrent`, `java.util.concurrent.atomic`

```
// Classe Dipendente dell'ufficio finanziario di facebook
package facebook.financeDepartment;

public class Dipendente { ... }
```

Package

I package possono essere annidati in altri package, permettendo di definire una **gerarchia di package**. L'annidamento di package è uno **strumento organizzativo** per package correlati, ma non fornisce alcun accesso particolare tra i package. Esempio dall'API: **`java.awt.image.renderable`** è un sottopackage di **`java.awt.image`** che è un sottopackage di **`java.awt`**, ma sono comunque dei package distinti.

I nomi di package gerarchici sono separati da punti, e sono convenzionalmente in minuscolo. Esempio: **`java.awt`**, **`java.awt.image`**, **`java.awt.image.renderable`**

La compilazione di un file sorgente con una dichiarazione di package **`nome1.nome2. ... .nomeK`** crea nella directory di compilazione **una gerarchia di sottodirectory**, se queste non esistono già, **`nome1/nome2/.../nomeK`** e in questa ultima sottodirectory viene generato il bytecode .class

Se un file .java non include una dichiarazione di package, il compilatore considera le classi di quel file come appartenenti ad un **package di default per la directory corrente**. Quindi tutte le classi senza dichiarazione di package compilate nella stessa directory appartengono allo stesso package di default

Esempio

```
// file C.java
package pack1;
public class C {}

// file D.java
package pack1.pack2;
public class D {}
```

```
/dir$ ls
C.java
D.java
/dir$ javac C.java D.java -d .
/dir$ ls . pack1/ pack1/pack2
.:
C.java*  D.java*  pack1/

pack1/:
C.class*  pack2/

pack1/pack2/:
D.class*
```

import

Il nome del package fa parte, come prefisso, del nome delle classi appartenenti al package. Ad esempio, si può usare **`facebook.financeDepartment.Dipendente`** per riferirsi, in qualsiasi classe, alla classe **`Dipendente`** del package **`facebook.financeDepartment`**, oppure **`java.awt.Button`** è il nome con il prefisso esplicito di package della classe **`Button`**

Si può evitare l'uso del prefisso esplicito del package di appartenenza mediante una **dichiarazione di `import`** in un file .java, che indica al compilatore in quali package cercare i nomi delle classi usate in quel file .java

```
// classe Dipendente del package facebook.financeDepartment
import facebook.financeDepartment.*;

public class Dirigente extends Dipendente {...}
```

La dichiarazione **`import pacchetto.*`** rende accessibile al file .java tutte le classi del package di nome **`pacchetto`**

Gli statement di **`import`** **devono precedere la dichiarazione di classe**

Qualora sia già presente una dichiarazione di appartenenza ad un package non è necessario importare lo stesso package o qualsiasi suo elemento

Lo statement di **`import`** non influenza in alcun modo la compilazione: non si tratta di una inclusione fisica di un file in un altro file, come accade per la direttiva **`#include`** del C++, ma semplicemente di una dichiarazione di un insieme di directory da utilizzare per risolvere i nomi di classe usati nel file

Variabile d'ambiente CLASSPATH

Quindi i file `.class` di una gerarchia di package vengono memorizzati nel file system in un albero di directory, dove ogni directory è un nome di package nella gerarchia. Come fare per rendere visibile la radice di queste directory?

`javac -d` *`directory`* specifica esplicitamente al compilatore una directory dove memorizzare i file `.class`, in particolare le classi di una gerarchia di package. Se non è specificata esplicitamente una directory, di default `javac` considera la directory corrente.

Le radici degli alberi di directory dei package possono essere specificate nella variabile d'ambiente `CLASSPATH`. Oppure l'invocazione della JVM specifica la radice mediante un flag:

```
java -cp /home/user/myprogram org.mypackage.HelloWorld
```

Controllo dell'accesso

I membri di una classe (campi dati e metodi) possono essere dichiarati con uno dei 4 livelli di accesso: **`public`**, **`protected`**, *default* (o *friendly*), **`private`**. Le classi possono essere dichiarate **`public`** o *default*. L'accesso di **default** è il cosiddetto ***accesso di package***

Le dichiarazioni sono di default quando non si mette alcun modificatore di accesso esplicito.

Modificatore	Stessa classe	Stesso package	Sottoclasse	Esterno
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	**Yes**	Yes	No
default	**Yes**	**Yes**	**No**	**No**
`private`	Yes	No	No	No

Esempio

```
// file D.java
// C e D nello stesso package di default

class C {
    protected int x=1;
}

public class D {
    public static void main(String[] args) {
       C c = new C();
       System.out.print(c.x); // compila e stampa 1
    }
}
```

Campi dati statici

Se si dichiara un campo dati **`static`**, tutti gli oggetti di quella classe condividono **un'unica copia condivisa** in memoria di quella variabile.

Java **non permette le variabili globali** (permesse invece dal C++).

Un **campo dati statico viene creato**, inizializzato automaticamente e tipicamente inizializzato esplicitamente **al momento del class loading.**

Come in C++ si accede ad un campo dati statico s di una classe C tramite il nome della classe però usando la seguente sintassi:

`C.s`

Campi dati statici: esempio

```
public class Count {
        public int serialNumber;
        private static int counter = 0;

        public Count() {counter++; serialNumber = counter;}
}
```

Al momento della creazione di un nuovo oggetto della classe `Count`, è assegnato un unico numero seriale nel campo `serialNumber`, sfruttando il campo dati statico `counter`, inizializzato a zero e incrementato di uno ad ogni nuova creazione di un oggetto

```
public class D {
        public void m() {
                Count c = new Count();
                System.out.println("Ho creato l'oggetto numero " +
                                c.serialNumber + " della classe Count");
                }
}
```

Campi dati statici

Campi dati statici si trovano in varie classi dell'API. Esempi: nella classe java.lang.Math abbiamo il campo dati statico **`public static final double PI`** per il numero pi-greco; nella classe java.lang.System, **`public static final PrintStream out`** è un campo dati statico di tipo PrintStream per lo stream di output standard

Field Summary

Fields

Modifier and Type	Field and Description
`static PrintStream`	**`err`** The "standard" error output stream.
`static InputStream`	**`in`** The "standard" input stream.
`static PrintStream`	**`out`** The "standard" output stream.

Class loading

Quando vengono caricate in memoria le classi?

Java è un linguaggio interpretato ed è quindi naturale che la politica per il class loading (cioè del bytecode di una classe) sia la seguente: *Il bytecode di una classe C viene caricato in memoria* ***solamente quando l'esecuzione richieda l'uso*** *della classe C.*

Quindi, il bytecode di una classe C è caricato in memoria dalla JVM al primo statement del bytecode che usa la classe C. Ad esempio, questo potrebbe essere la costruzione di un oggetto di C tramite una new oppure l'accesso ad un campo dati o metodo statico di C.

Attenzione: la semplice **dichiarazione di un reference** di tipo C è uno statement che non necessita del caricamento del bytecode di C.

Esempio

```
class A {
    String s;
    A() {s="pippo"; System.out.println(s);}
    A(String x) {s=x; System.out.println(s);}
}
class C {
    static A s1;
    static A s2 = new A("pluto");
}
public class E {
  public static void main(String[] args) {
    System.out.println("Inizio E.main()");
    C c;
    System.out.println("Dopo C c;");
    System.out.println(C.s1 + " " + C.s2);
    // System.out.println(C.s1.s);
    // run time exception: NullPointerException
    System.out.println(C.s2.s);
    System.out.println("Fine E.main()");
  }
}
```

caricamento di C

```
/* STAMPA:
Inizio E.main()
Dopo C c;
pluto
null A@3ecf72fd
pluto
Fine E.main()
*/
```

Metodi statici

Come al solito, i metodi statici, cioè marcati **`static`**, vengono usati per implementare funzionalità indipendenti da una particolare istanza della classe, e quindi tipicamente codificano azioni specifiche della classe.

Un metodo statico **`m()`** di una classe **`C`** è invocato dall'esterno della classe **`C`** con la sintassi: **`C.m();`**

Metodi statici

```
public class FunzioniVarie {
    public static int add(int x, int y) {return x+y;}
}
public class Usa {
    public void stampaSomma(int m, int n)
        System.out.println(m + " + " + n + " = " + Funzioni.add(m,n));
    }
}
```

- **Naturalmente:** in un metodo statico **non** è a disposizione il riferimento **`this`**.
- Il metodo **`main()`** è sempre **`static`** e **`public`** perchè deve poter essere accessibile per far iniziare l'esecuzione prima che qualsiasi istanza della classe contenente il **`main()`** sia stata creata.

Blocchi statici

Una classe può contenere un cosiddetto ***blocco statico*** contenente del codice che non appare in alcun corpo di metodo. Il codice nel blocco statico viene eseguito **una sola volta al momento del caricamento della classe**. Se sono presenti più blocchi statici, essi vengono eseguiti sequenzialmente seguendo l'ordine testuale della loro definizione nella classe. La sintassi è descritta dal seguente esempio

```
class Static {
    public static int n = 3;
    static {
        System.out.println("Codice statico: n = " + n); n++;
    }
}
public class Usa {
    public static void main(String args[]) {
        System.out.println("Codice main: n = " + Static.n);
    }
}
/* STAMPA:
Codice statico: n = 3
Codice main: n = 4
*/
```

```
public class Car {
    private String modello;
    private String colore;
    // Numero di serie specifico di ogni istanza
    private int numeroDiSerie;
    // statico: accessibile da tutte le istanze
    private static int prossimoNumeroDiSerie = 1;
    public Car(String modello, String colore) {
          this.modello = modello; // bisogna usare this per disambiguare
          this.colore = colore; numeroDiSerie = prossimoNumeroDiSerie++;
    }
    public static void stampaProssimoNumeroDiSerie() {
          System.out.println("Il prossimo numero di serie disponibile è " +
                                  prossimoNumeroDiSerie);
    }
    public void stampaDati() {
          System.out.println("Questa è una " + modello + " " + colore +
                                  ", numero di serie: " + numeroDiSerie);
    }
    public static void main(String args[]) {
          Car pippoCar = new Car("Porsche Carrera", "grigia");
          Car plutoCar = new Car("Ferrari Testarossa", "gialla");
          pippoCar.stampaDati(); plutoCar.stampaDati();
          // invoco metodi statici per ottenere informazioni di classe
          stampaProssimoNumeroDiSerie();
    }
}
```

```
class D {
  int i = 4;
  int j;
  D() {
    print("i = " + i + ", j = " + j);
    j = 7;
  }
  static int x1 = print("static D.x1");
  static int print(String s) {
    System.out.println(s);
    return 9;
  }
}

public class E  {
  int k = D.print("non static E.k");
  E() { D.print("k = " + k); }
  static int x2 = D.print("static E.x2");

  public static void main(String[] args) {
    D.print("invocazione di E()");
    E e = new E();
  }
}
```

```
/* STAMPA:
static D.x1
static E.x2
invocazione di E()
non static E.k
k = 9
*/
```

```
class D {
  int i = 4;
  int j;
  D() {
    print("i = " + i + ", j = " + j);
    j = 7;
  }
  static int x1 = print("static D.x1");
  static int print(String s) {
    System.out.println(s);
    return 9;
  }
}

public class E extends D {
  int k = D.print("non static E.k");
  E() { D.print("k = " + k); }
  static int x2 = D.print("static E.x2");

  public static void main(String[] args) {
    D.print("invocazione di E()");
    E e = new E();
  }
}
```

```
/* STAMPA:
static D.x1
static E.x2
invocazione di E()
non static E.k
k = 9
*/
```

```
/* STAMPA:
static D.x1
static E.x2
invocazione di E()
i = 4, j = 0
non static E.k
k = 9
*/
```

Blocchi statici e campi statici

Cosa succede al momento del caricamento di una classe che contiene sia campi dati statici che blocchi statici? Vengono prima eseguiti i blocchi statici o vengono prima inizializzati i campi dati statici? Vediamo un esempio

Blocchi statici e campi statici

```
class Z {
   public Z(int x) {k=x; System.out.println("Z(" +x+ ")");}
   public int k;
}
class A {
   static Z z = new Z(1);
   static {System.out.println("blocco statico A");}
}
class B {
   static {System.out.println("blocco statico B");}
   static Z z = new Z(2);
}
public class C {
   public static void main(String args[]) {
         A a = new A();
         System.out.println();
         B b = new B();
   }
}
```

Blocchi statici e campi statici

```
class Z {
   public Z(int x) {k=x; System.out.println("Z(" +x+ ")");}
   public int k;
}
class A {
   static Z z = new Z(1);
   static {System.out.println("blocco statico A");}
}
class B {
   static {System.out.println("blocco statico B");}
   static Z z = new Z(2);
}
public class C {
   public static void main(String args[]) {
         A a = new A();
         System.out.println();
         B b = new B();
   }
}
```

```
/* STAMPA:
Z(1)
blocco statico A

blocco statico B
Z(2)
*/
```

Quindi viene sempre seguito l'**ordine di dichiarazione nella classe dei campi dati statici e dei blocchi statici**, a meno di "illegal forward references" che appunto sono illegali (si veda il prossimo esempio)

Blocchi statici e campi statici

```
class Z {
   public Z(int x) {k=x; System.out.println("Z(" +x+ ")");}
   public int k;
}
class A {
   static Z z = new Z(1);
   static {System.out.println("blocco statico A");}
}
class B {
   static {System.out.println("blocco statico B"); (z.k)++;}
   static Z z = new Z(2);
}
public class C {
   public static void main(String args[]) {
         A a = new A();
         System.out.println();
         B b = new B();
   }
}
```

```
/* NON COMPILA
z is an illegal
forward reference
*/
```

Sui reference

Attenzione: quando si usa come r-value (cioè in lettura) un reference in un qualsiasi statement, questo deve sempre essere stato esplicitamente inizializzato, eventualmente a null, altrimenti il compilatore segnala che il reference "might not have been initialized".

```
class A { public void m(Object obj) {System.out.println("A.m(Object)");} }

class B extends A { public void m(Object obj) {
                                        System.out.println("B.m(Object)"); }
}
public class C extends B  {
    public void m(Object obj) {System.out.println("C.m(Object)");}
    public static void main(String[] args) {
         A a; B b; C c = new C();
        a.m(c); // ILLEGALE: variable a might not have been initialized
        a=c; a.m(c); // OK, stampa: C.m(Object)
        a.m(b); // ILLEGALE: variable a might not have been initialized
        a=null; a.m(c); // Eccezione: NullPointerException
    }
}
```

L'operatore ==

- L'operatore **==** esegue un confronto di uguaglianza tra il valore di **variabili di uno stesso tipo qualsiasi**, primitivo o definito da utente
- **Attenzione:** in un confronto **`x==y`**, le variabili **`x`** e **`y`** devono essere state inizializzate altrimenti lo statement non compila (a variable might not have been initialized)
- In particolare, se **`x`** e **`y`** sono **reference**, cioè di tipo non primitivo, **`x==y`** ritorna **`true`** **se e solo se** **`x`** e **`y`** si **riferiscono allo stesso oggetto** (ovvero memorizzano lo stesso indirizzo)
- **Attenzione:** `x=null; y=null; System.out.print(x==y);` compila correttamente e stampa true

Il metodo `equals()`

La classe `Object` include il metodo

`public boolean equals(Object)`

`equals()` può quindi **sempre essere invocato** su qualsiasi oggetto. Nell'implementazione della classe `Object`, `ref1.equals(ref2)` ritorna `true` se e solo se i due reference `ref1` e `ref2` si riferiscono allo stesso oggetto. Quindi, `equals(Object)` è implementato in `Object` come l'operatore ==. La differenza tra == ed equals è quindi data dalla **tipizzazione**

```
class B {}      class C extends B {}      class D {}

public class E  {
  public static void main(String[] args) {
      B b = new B(); C c = new C(); D d = new D();
      System.out.println(b==c);         // stampa: false
      System.out.println(b.equals(c)); // stampa: false
      // System.out.println(b==d);      // Non compila
      System.out.println(b.equals(d)); // stampa: false
  }
}
```

Il metodo `equals()`

Tipicamente quando si invoca il metodo `equals()` si vorrebbero **confrontare gli oggetti puntati** dai reference e non i reference. Infatti di prassi il metodo `equals()` è ridefinito in una classe proprio con questo significato. Ad esempio, la classe dell'API java.util.Date (e molte altre classi dell'API tra cui String, File, ... le classi wrapper che vedremo nel seguito) ridefinisce this.equals(Object ref) in modo che ritorni true se e soltanto se i due oggetti puntati dai due reference this e ref hanno come tipo dinamico un sottotipo di Date (cioè TD(this) ≤ Date e TD(ref) ≤ Date) e i due sottooggetti di this e ref di tipo Date rappresentano la stessa data.

`java.util.Date`

```
public boolean equals(Object obj)
```

Compares two dates for equality. The result is `true` if and only if the argument is not `null` and is a `Date` object that represents the same point in time, to the millisecond, as this object.

Thus, two `Date` objects are equal if and only if the `getTime` method returns the same `long` value for both.

Overrides:

`equals` in class `Object`

```
public class E { // E non ridefinisce equals()
      private String s;
      public E(String o) {s=o;}

    public static void main (String args []) {
      E ref1 = new E("Gastone"), ref2 = new E("Gastone");
      if(ref1.equals(ref2)) System.out.println("ref1 equals ref2");
      else System.out.println("!(ref1 equals ref2)");
      String s1 = new String("Gastone"), s2 = new String("Gastone");
      // La classe String ha ridefinito equals()
      if(s1.equals(s2)) System.out.println("s1 equals s2");
      else System.out.println("!(s1 equals s2)");
    }
}
/* STAMPA:
!(ref1 equals ref2)
s1 equals s2
*/
```

```
import java.util.*;

class C extends Date {
    long i;
    public C(long n) {super(n); i=n;}
}
class D extends Date {
    long j;
    public D(long n) {super(n); j=n+n;}
}
public class E  {
  public static void main(String[] args) {
      Date d = new Date(1), e = new Date(1), x = new Date(2);
      C y = new C(2); D z = new D(2);
      System.out.println(d.equals(e)); // stampa: true
      System.out.println(x.equals(y)); // stampa: true
      System.out.println(y.equals(d)); // stampa: false
      System.out.println(y.equals(z)); // stampa: true
  }
}
```

Il metodo `toString()`

Abbiamo già visto che `String toString()` è un metodo della classe `Object`, ed è quindi ereditato da qualsiasi classe. L'implementazione di `toString()` in `Object` ritorna una stringa contenente il **nome della classe e l'hashcode** contenuto nel reference di invocazione.

La prassi di ridefinizione del metodo `toString()` è quella di **convertire un oggetto in una rappresentazione stringa** in qualche modo significativa ed informativa. Abbiamo inoltre già visto che il compilatore inserisce delle invocazioni implicite al metodo `toString()` in ogni contesto dove sia usato un reference e si richieda invece una espressione di tipo `String`

```
Data d = new Data();
System.out.println(d);
// il compilatore lo traduce nella chiamata
// System.out.println(d.toString())
```

Keyword `final`

La keyword final ha significati leggermente diversi a seconda del contesto. Il significato generale è: "**Questo non può essere cambiato**". E' una sorta di "const" per Java. Le motivazioni per evitare un cambiamento possono essere di due tipologie: **progettazione ed efficienza**. Queste due motivazioni sono piuttosto diverse tra di loro ed è quindi possibile fare un cattivo uso della keyword final. La keyword final si può usare per i campi dati, gli argomenti di metodi, i metodi e le classi.

Il reference `this` è `final`

Il reference **`this`**, che in un metodo non statico si riferisce all'oggetto di invocazione, è sempre implicitamente final, ovvero non è modificabile.

Attenzione: l'oggetto a cui **`this`** si riferisce ovviamente è invece modificabile.

```
public class C {
  public int x;
  public void m() {
    // this = new C();
    // ILLEGALE: cannot assign a value to final variable this
    this.x=5; // OK
}
```

Campi dati `final`

Un campo dati, sia statico che di istanza, può essere marcato come final. Ciò significa che in realtà quel **campo dati è costante**: **non può essere modificato**, ovvero ogni tentativo di modificarne il valore è illegale (non viene compilato).

I **campi dati statici finali** sono anche detti *costanti di classe* e **devono necessariamente essere inizializzati esplicitamente**.

I **campi dati di istanza finali** solitamente sono inizializzati esplicitamente, ma ciò non è obbligatorio: se non sono inizializzati esplicitamente ogni costruttore deve necessariamente inizializzarli, ossia non basta l'inizializzazione automatica.

Campi dati `final`

```
public class E  {
  // final int i; // ILLEGALE: manca il costruttore
}

public class E {
  final int i;
  E() {i=0;} // OK
}

public class E {
  final int i;
  E() {i=0; /* i=2; */} // ILLEGALE
}

public class E {
  // static final int k; // ILLEGALE
  static final int k=0;  // OK
}
```

Campi dati `final`

Attenzione: se un reference ref viene marcato final, allora ref si riferisce ad uno ed uno solo oggetto fissato, ma l'oggetto stesso può comunque essere modificato. A differenza di C++, **Java non fornisce un modo per rendere costanti gli oggetti** a cui si riferisce un reference.

I campi dati reference final sono raramente utilizzati.

Campi dati `final`

```
class Z {
    int i=2;
}
public class D {
  Z z1 = new Z();
  final Z z2 = new Z();
  static final Z z3 = new Z();
  final int[] a = { 1, 2, 3, 4, 5, 6 };

  public static void main(String[] args) {
    D d1 = new D();
    d1.z2.i++; // OK
    d1.z1 = new Z(); // OK
    for(int i = 0; i < d1.a.length; i++)
      d1.a[i]++; // OK, solamente il reference a è costante
    // d1.z2 = new Z();   // ILLEGALE
    // D.z3 = new Z();    // ILLEGALE
    // d1.a = new int[3]; // ILLEGALE
  }
}
```

Argomenti `final`

È possibile marcare gli **argomenti di un metodo come final**. Questo significa che nel corpo del metodo non è possibile modificare l'argomento, sia esso di tipo primitivo oppure un reference

```
class Z {
    int i=2;
}
public class D {
    public void m(final Z ref) {
        // ref = new Z(); // ILLEGALE
        ref.i=3; // OK
    }
    public int m(final int i) {
        // i++; // ILLEGALE
        return i+1; // OK
    }
}
```

Metodi `final`

Un metodo può essere marcato come final: ciò significa che il metodo **non può essere ridefinito in eventuali sottoclassi**. Se si tenta di farlo il compilatore segnala una illegalità.

Metodi `final`

Sono due le motivazioni per marcare un metodo final

1) Per **motivi progettuali** si desidera mettere un "blocco" sul metodo per evitare che una sottoclasse possa cambiarne il comportamento. Si pensi a motivazioni di “security”: se l’implementazione di un metodo final non fa uso di (cioè non invoca) metodi non-finali, il suo comportamento è completamente fissato e immodificabile. Ad esempio, si può pensare che un metodo `boolean validatePassword()` debba essere marcato `final`, per evitare, ad esempio, ridefinizioni "maliziose" che restituiscano sempre `true`
2) Un metodo può venire marcato final anche per **motivi di efficienza**. Infatti quando si marca come finale un metodo il compilatore genera una chiamata statica a quel metodo finale invece di utilizzare il late binding dei metodi virtuali. Quindi una chiamata ad un metodo final è più efficiente in quanto evita l'overhead del late binding

Metodi `final`

Quando il compilatore trova una chiamata ad un metodo final può **a sua discrezione** (non c'è garanzia che succeda effettivamente) sostituire quella chiamata con una **copia in-line** del codice del metodo final.

```
class C {
    public final void m() {System.out.println("m() di C");}
}
public class D extends C {
    public void m() {System.out.println("m() di D");}
}
/*  m() in D cannot override m() in C; overridden method is final */
```

Metodi `final`

```
class B {
    public final void f() {for(int i=0; i<10; ++i);} // finale
    public void g() {for(int i=0; i<10; ++i);} // "virtuale"
}
class C extends B {public void g() {for(int i=0; i<10; ++i);}}
class D extends C {public void g() {for(int i=0; i<10; ++i);}}

public class Test {
  public static void main(String[] args) {
    B[] a = new B[50000000];
    for(int i=0; i<50000000; ++i) {
      if(i%3==0) a[i] = new B(); if(i%3==1) a[i] = new C();
      if(i%3==2) a[i] = new D();
    }
    long inizio = System.currentTimeMillis();
    //for(int i=0; i<a.length; ++i) a[i].f(); // chiamata statica
    //for(int i=0; i<a.length; ++i) a[i].g(); // chiamata dinamica
    System.out.println(System.currentTimeMillis()-inizio);
    }
}
// nella mia JVM differenza di circa 84%
```

Metodi `final`

Tutti i metodi marcati **`private`** di una classe sono implicitamente **`final`**. Infatti poichè non è possibile accedere ad un metodo private non è nemmeno possibile fare l'overriding di quel metodo. Anche se si aggiunge la keyword **`final`** ad un metodo privato ciò non conferisce alcun significato addizionale al metodo.

Tuttavia se si tenta di "**ridefinire**" (con la stessa segnatura) un **metodo privato** m() in una sottoclasse il compilatore non segnala una illegalità: infatti in tal modo semplicemente si definisce un **nuovo metodo** nella sottoclasse e non si fa overriding.

```
class B {
  // final superfluo
  private final void f() { System.out.println("B.f()"); }
  private void g() { System.out.println("B.g()"); }
}
class C extends B {
  private final void f() { System.out.println("C.f()"); }
  public void g() { System.out.println("C.g()"); }
}
class D extends C {
  public final void f() { System.out.println("D.f()"); }
  public void g() { System.out.println("D.g()"); } // overriding
}
public class E {
  public static void main(String[] args) {
      D d = new D();
      d.f(); // stampa: D.f()
      d.g(); // stampa: D.g()
      C c = d; // upcast
      // c.f(); // ILLEGALE: f() has private access in C
      c.g(); // OK, stampa: D.g()
      B b = d; // upcast
      // b.f(); // ILLEGALE: f() has private access in B
      // b.g(); // ILLEGALE: g() has private access in B
  }
}
```

Classi `final`

Una intera classe C può essere marcata come final: ciò significa che **non possono essere definite sottoclassi** di C.

Sono esempi di classi final dell'API: `java.lang.String`, `java.lang.Math` e `java.lang.System`

Le **motivazioni** per marcare una classe final sono spesso di "**security**". Ad esempio, un reference a `String` sicuramente si riferisce ad un oggetto di tipo `String`, e non ad una "stringa" di qualche sottoclasse di `String` che potrebbe essere un tipo "malizioso".

Quindi se ref è un reference tale che TS(ref)=C e C è una classe final allora sicuramente TD(ref)=C.

Naturalmente, tutti i metodi di una classe final sono implicitamente final.

Marcando una classe come final si opera una notevole restrizione al suo utilizzo, dal momento che si restringe la flessibilità della classe per eventuali estensioni che volessero aumentare le funzionalità della classe. Le **motivazioni progettuali** che spingono a marcare final una intera classe devono quindi essere **piuttosto forti**.

Classi astratte

È un concetto generale OO già approfondito con il C++. Una classe è astratta quando dichiara l'esistenza di metodi ma non definisce la loro implementazione. In Java, si usa il **marcatore `abstract` sia per la classe che per i metodi**.

Tipicamente, una classe astratta fornisce una intelaiatura corrispondente a funzionalità di tipo generale, richiedendo poi la definizione delle funzionalità specifiche mancanti alle sottoclassi. I metodi dichiarati ma non implementati devono necessariamente essere marcati `abstract` (il compilatore segnala un errore se una classe non è marcata abstract ed invece contiene un metodo marcato abstract). Naturalmente, le classi astratte possono avere, e molto spesso hanno, dei campi dati.

Classi astratte: regole

- **Non è possibile istanziare oggetti di una classe astratta**: se A è astratta uno statement "new A(...);" viene segnalato come un errore dal compilatore
- È invece **possibile avere costruttori astratti**: anzi, in generale ci saranno dei costruttori quando ci sono dei campi dati nella classe astratta. I costruttori verranno richiamati dai costruttori delle classi che derivano dalla classe astratta.
- Una sottoclasse di una classe astratta A è **concreta** quando fornisce le implementazioni di **tutti** i metodi astratti di A. Se non lo fa, anche tale sottoclasse è astratta
- In una classe astratta ci possono essere dei **metodi** implementati **che invocano metodi astratti**: a run-time verrà sempre invocato dinamicamente l'opportuno metodo concreto (se esiste, altrimenti verrà sollevata a run-time un'eccezione)
- È possibile, ovviamente, dichiarare un **reference il cui tipo (statico) è una classe astratta** A: il tipo dinamico di un tale reference dovrà necessariamente essere un sottotipo proprio di A. Quindi una classe astratta può essere solamente un tipo statico, mai un tipo dinamico
- **Non è possibile dichiarare metodi statici astratti**
- Chiaramente **non è possibile avere un metodo final astratto**: sarebbe un "non-sense"
- A differenza del C++, è possibile definire una **classe astratta ma senza alcun metodo astratto**: ciò può succedere qualora il progetto richieda che non sia possibile istanziare quella classe

Classi astratte: esempio

```
public abstract class Drawing {
    public abstract void drawDot(int x, int y);

    public void drawHLine(int x1, int x2, int y) {
       // disegna una linea usando il metodo astratto drawDot()
       for(int x=x1; x<=x2; x++) drawDot(x,y);
    }
}
public class LinuxDrawing extends Drawing {
    public void drawDot(int x, int y) { // implementazione Linux
          ...
    }
}
public class MacOSDrawing extends Drawing {
    public void drawDot(int x, int y) { // implementazione MacOS
          ...
    }
}
void someMethod(){
    Drawing dLinux = new LinuxDrawing(); dLinux.drawLine(1,1,3,3);
    Drawing dMacOS = new MacOSDrawing(); dMacOS.drawLine(1,1,3,3);
    ...
}
```

Classi astratte: esempio

```
public abstract class A {
  private int a;
  public A(int x) {a=x;}
  public abstract int m();
}

public class C extends A { // C è una concretizzazione di A
  private int b;
  public int m() {return 4;} // implementazione di m()
  public C() {super(8); b=0;} // OK
  //public C(int x){} // ILLEGALE: No constructor matching A() found in class A

  public static void main(String args[]) {
    // A r = new A(9); // ILLEGALE: Abstract class A can't be instantiated
    A s = new C(); // OK
  }
}
```

Classi astratte: esempio

```
abstract class A {
    public abstract void m();
}
class B extends A {
    public void m() { System.out.println("B.m()");}
}
class C extends A {
    public void m() { System.out.println("C.m()");}
}
class D extends A {
    public void m() { System.out.println("D.m()"); }
}
public class E {
    // per f() e g() nulla cambia se la gerarchia
    // viene estesa con nuove sottoclassi concrete di A
    static void f(A ref) { ref.m(); }
    static void g(A[] a) { for(int i = 0; i < a.length; i++) f(a[i]); }
    public static void main(String[] args) {
      A[] a = {new B(), new C(), new D(), new C()};
      g(a); // stampa: B.m() C.m() D.m() C.m()
    }
}
```

Esercizio cavillo

Abbiamo visto che quando viene costruito un oggetto di una classe derivata D con una chiamata ad un costruttore succede che nell'ordine: (1) viene chiamato un costruttore, esplicitamente o implicitamente, per la classe base diretta di D; (2) viene allocata la memoria per i campi dati propri di D e viene inizializzata automaticamente; (3) vengono eseguite le eventuali inizializzazioni esplicite di D; (4) viene eseguito il corpo del costruttore di D.

In effetti ciò non è del tutto esatto. Consideriamo il seguente esempio in cui effettuiamo una chiamata polimorfa in un costruttore di una classe base astratta. Compila? Viene lanciata un'eccezione a run-time? Se l'esecuzione termina cosa stampa?

Esercizio cavillo

```
abstract class A {
    public abstract void m();
    public A() {
         System.out.println("In A() prima di m()");
         m(); // chiamata polimorfa
         System.out.println("In A() dopo di m()");
    }
}
class B extends A {
    int k = 1;
    public void m() { System.out.println("B.m(), campo dati k = " + k); }
    public B(int x) {k=x; System.out.println("B.B(), campo dati k = " + k); }
}
public class C {
    public static void main(String[] args) {
         new B(3);
    }
}
```

A rigore, quando avviene la chiamata m() dentro A(), verrà invocato m() di B ed il campo dati k di B non dovrebbe ancora essere stato allocato. Invece compila e stampa:

Esercizio

```
abstract class A {
    public abstract void m();
    public A() {
        System.out.println("In A() prima di m()");
        m(); // chiamata polimorfa
        System.out.println("In A() dopo di m()");
    }
}
class B extends A {
    int k = 1;
    public void m() { System.out.println("B.m(), campo dati k = " + k); }
    public B(int x) {k=x; System.out.println("B.B(), campo dati k = " + k); }
}
public class C {
    public static void main(String[] args) {
        new B(3);
    }
}
```

A rigore, quando avviene la chiamata m() dentro A(), verrà invocato m() di B ed il campo dati k di B non dovrebbe ancora essere stato allocato. Invece compila e stampa:

```
/* STAMPA:
In A() prima di m()
B.m(), campo dati k = 0
In A() dopo di m()
B.B(), campo dati k = 3
*/
```

Esercizio cavillo

Spiegazione:

Quando viene costruito un oggetto di una classe derivata D tramite la chiamata ad un costruttore: (1) viene innanzitutto allocata la memoria ed inizializzata automaticamente a zero per tutti i campi dati di D, sottoggetto incluso; (2) viene chiamato un costruttore per la classe base diretta di D; (3) vengono eseguite le eventuali inizializzazioni esplicite per i campi dati di D; (4) viene eseguito il corpo del costruttore

Interfacce

Le *interfacce* (keyword **`interface`**) sono una variazione dell'idea di classe astratta. Le interfacce offrono un modo per dichiarare dei tipi formati solamente da costanti e metodi astratti, permettendo di scrivere diverse implementazioni per tali metodi. Quindi, un'interfaccia si può vedere come una **definizione puramente progettuale** di un tipo che fornisce degli obiettivi generali che ogni sua implementazione deve perseguire. In altri termini, un'interfaccia è utilizzata per stabilire un protocollo tra classi: un'interfaccia dichiara delle garanzie riguardo ai servizi che possono essere richiesti ad un determinato oggetto. Si può anche dire che un'interfaccia è una *"classe astratta pura".*

In un'interfaccia tutti i metodi sono **implicitamente abstract**, e quindi nessun metodo può essere definito. In particolare, non possono essere presenti metodi statici in un'interfaccia, visto che un metodo statico non può essere abstract. I **metodi** di un'interfaccia inoltre **sono sempre pubblici** ed il marcatore **`public`** può essere omesso. Un'interfaccia può avere solamente campi dati statici e final, quindi solamente delle costanti.

```
public interface Transparency {
    public static final int OPAQUE = 1;
    public static final int BITMASK = 2;
    public static final int LUCENT = 3;
    public void setTransparency(int level);
    public int getTransparency();

}
```

Interfacce

Per definire una classe che **implementa un'interfaccia o un insieme di interfacce** si utilizza la keyword **`implements`**. Tale classe deve definire **tutti i metodi** dell'interfaccia: se non lo fa, il compilatore segnala un'illegalità. Una classe che implementa un'interfaccia diventa una classe ordinaria che può essere estesa in modo ordinario.

```
public interface Hello { public void hello(); }
public interface Ciao { public void ciao(); }

public class HelloImpl implements Hello {
    public void hello() { System.out.println("Hello"); }
}
public class HelloCiaoImpl implements Hello, Ciao {
    public void hello() { System.out.println("Hello"); }
    public void ciao() { System.out.println("Ciao"); }
}
```

Interfacce

Le interfacce possono anche essere estese da altre interfacce usando la keyword **`extends`**. Diversamente dalle classi, **un'interfaccia può estendere più di un'interfaccia**. Si può parlare di **multiple subtyping** piuttosto che **multiple inheritance**

```
public interface Hello { public void hello(); }
public interface Ciao extends Hello { public void ciao(); }

public class CiaoImpl implements Ciao {
    public void hello() { System.out.println("Hello"); }
    public void ciao() { System.out.println("Ciao"); }

}
```

Interfacce

È possibile usare **un'interfaccia come tipo statico per un reference**, ovvero dichiarare reference che hanno come tipo un'interfaccia. Per tali reference, ci sarà a run-time il late binding per l'invocazione dei metodi. Quindi le **interfacce sono dei tipi** a tutti gli effetti. Ovviamente non è possibile costruire oggetti di un'interfaccia. È possibile fare dei cast verso o da tipi interfaccia. Ed è quindi possibile usare l'operatore **`instanceof`** per riferimenti ad interfacce. Non esiste una interfaccia analoga all'Object delle classi. Vale comunque il subtyping su Object, cioè è comunque possibile assegnare ad un reference obj di tipo Object un reference ref che abbia come tipo statico un'interfaccia in quanto ref dovrà essere inizializzato, cioè riferirsi ad un oggetto di qualche classe C che naturalmente è una sottoclasse di Object.

```
class Z {}      interface I {}

public class C implements I  {
    public static void m(I ref) {Object o = ref; System.out.println("m(I)");}
    public static void m(Object obj) {System.out.println("m(Object)");}
    public static void main(String[] args) {
        C c = new C(); I i =c; Z z = new Z();
        m(i); // OK, stampa: m(I)
        m(c); // OK, stampa: m(I)
        m(z); // OK, stampa: m(Object)
    }
}
```

Interfacce ed ereditarietà

Le interfacce permettono di superare in modo controllato il vincolo Java dell'ereditarietà singola: mentre una classe può **estendere** un'**unica** singola classe, può invece **implementare un numero qualsiasi di interfacce**. Inoltre, una classe può simultaneamente estendere una classe ed implementare alcune interfacce (sintassi: prima deve comparire la keyword extends e poi implements).

```
import java.awt.*;
import java.lang.*;
import java.awt.event.*;

public class MyFrame extends Frame
                     implements MouseListener, MouseWheelListener {
...
}
```

Interfacce ed ereditarietà

```
interface I {
    void m();
}
interface J {
    void n();
}
class B {
    public void m() {System.out.println("B.m()");}
}
// C non fornisce implementazione di I.m() perchè è ereditata da B
class C extends B implements I, J  {
    public void n() {System.out.println("C.n()");}
}
public class D {
    public static void f(I x) {x.m();}
    public static void g(J x) {x.n();}
    public static void h(B x) {x.m();}
    public static void main(String[] args) {
          C c = new C();
          f(c); // OK, upcast a I, stampa: B.m()
          g(c); // OK, upcast a J, stampa: C.n()
          h(c); // OK, upcast a B, stampa: B.m()
    }
}
```

Interfacce ed ereditarietà

Il multiple subtyping con le interfacce elimina i problemi connessi all'ereditarietà multipla del C++. Infatti in C++ si presenta il problema dell'ambiguità per una derivazione di classi a diamante che motiva l'introduzione delle classi virtuali: A è una classe base contenente un metodo m(), possibilmente e tipicamente metodo virtuale, le classi B e C derivano da A ed entrambe ridefiniscono m(), D deriva sia da B che da C, ha quindi due sottooggetti di tipo A e conseguentemente per un puntatore D* pd ad un oggetto di tipo D una chiamata pd->m() viene segnalata dal compilatore come ambigua. In Java tale problema non sussiste.

Dimostrazione: Se D è una classe sottotipo diretto sia di B che di C che sono sottotipi diretti di A allora o B o C deve essere un'interfaccia. Di conseguenza anche A dovrà necessariamente essere un'interfaccia. Quindi m() è solamente dichiarato in A. Se B e C sono entrambe interfacce allora m() è implementato in D. Se invece una tra B e C è una classe, ad esempio B, allora m() è implementato in B, e possibilmente ridefinito in D. In tutti i casi se d è un reference di tipo D, la chiamata d.m() non genera alcuna ambiguità.

Cavilli sul tipo di ritorno

```
interface I { void m(); }
class A {
    public int m() {return 1;}
}
public class E extends A implements I {}
```

```
/* NON COMPILA:
m() in A cannot implement m()
in I; attempting to use
incompatible return type
*/
```

Cavilli sul tipo di ritorno

```
interface I { void m(); }
interface J { int m(); }
interface K extends I, J {}
```

```
/* NON COMPILA:
interfaces J and I are
incompatible; both define
m(), but with different
return type
*/
```

Utilità delle interfacce

- Permettono di rendere pubblica solo "l'interfaccia" di programmazione di una classe, senza dare alcun dettaglio dell'effettiva implementazione, ovvero possono essere viste come un "puro contratto"
- Permettono di catturare le similitudini tra classi apparentemente non correlate, senza forzare una relazione di sottoclasse
- Permettono di fare upcast a più di un tipo base, grazie al multiple subtyping

Interfacce vs classi astratte

- Alcuni esperti OO sostengono che ogni classe importante, che presumibilmente potrebbe essere estesa, dovrebbe implementare un'interfaccia
- **Vantaggio:** Se si definisce una classe C invece di un'interfaccia I che dichiara i metodi pubblici di C e quindi implementata da una classe CI "equivalente" a C (cioè che implementa i metodi di I come in C), allora non sarà possibile definire nuove classi che estendano sia C che altre classi esistenti. Invece, con l'interfaccia I e l'implementazione CI è possibile scegliere di estendere direttamente la classe CI oppure di implementare l'interfaccia I e contemporaneamente estendere un'altra classe: in questo secondo modo, I riveste il ruolo del "tipo di C" e quindi riusciamo ad ottenere un nuovo tipo che è sottotipo "sia di C" che di un altro tipo.
- **Interfaccia o classe astratta?** Un'interfaccia offre i vantaggi di una classe astratta e anche i benefici di un'interfaccia, quindi se il contesto permette di definire il tipo base senza implementare alcun metodo si dovrebbe preferire un'interfaccia ad una classe astratta. Tuttavia, il fatto che una classe astratta permetta di fornire alcune implementazioni particolarmente importanti che possono quindi essere ereditate può a volte essere utile e vantaggioso rispetto all'impiego dei rinvii espliciti per le implementazioni delle interfacce.

Subtyping

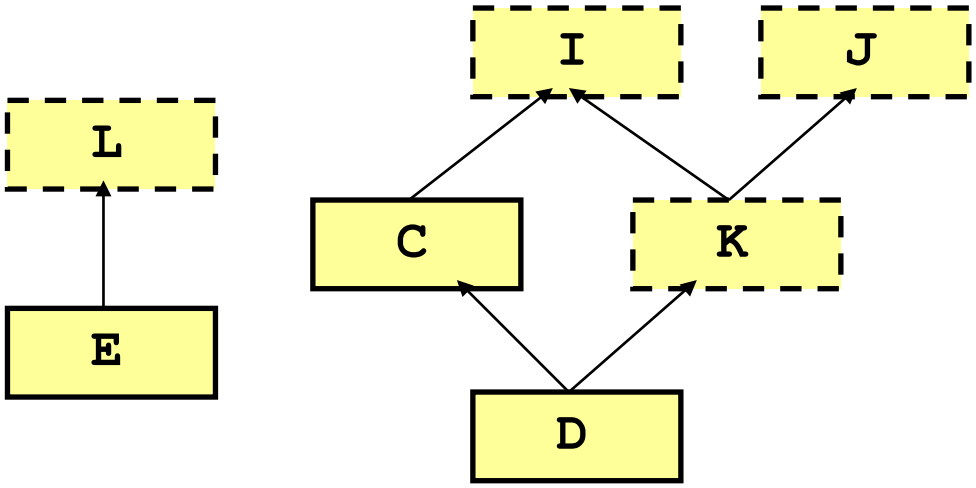
Aggiorniamo la relazione $\leq$ di subtyping includendo anche le interfacce (e le classi astratte) tra i tipi (si ricordi che interfacce e classi astratte possono agire solamente da tipo statico per un reference). Siano T1 e T2 due qualsiasi tipi (classe o interfaccia). Allora T1 $\leq$ T2 se vale una delle seguenti condizioni:

1) T1 e T2 sono lo stesso tipo (classe o interfaccia)
2) T1 è definito come (dove N $\geq$ 0 e [...] significa opzionale)
   a) class **T1** extends **T2** [implements Type1,...,TypeN]
   b) class **T1** [extends Type] **implements** Type1, ..., **T2**, ..., TypeN
   c) interface **T1** extends Type1, ..., **T2**, ..., TypeN
3) T1 è un tipo array T1a[], T2 è un tipo array T2a[] e vale T1a $\leq$ T2a
4) Esiste un tipo T3 tale che T1 $\leq$ T3 e T3 $\leq$ T2
5) T2 è il tipo `Object`

Cast

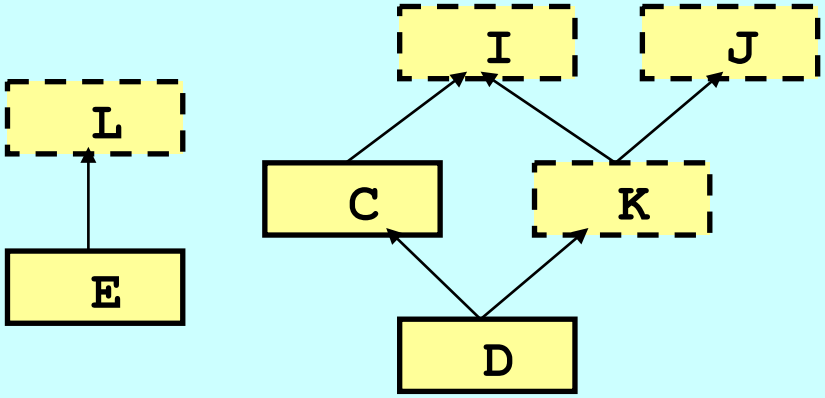
Con l'introduzione delle interfacce la relazione di subtyping permette di definire delle gerarchie in cui un tipo ha due (o più) supertipi diretti distinti

Ad esempio possiamo avere la seguente situazione:



```
interface I { }    interface J { }    interface K extends I,J { }    interface L { }
class C implements I { }              class E implements L { }

public class D extends C implements K {
    private static void f1() {
        D d = new D(); C ref = new D();
        I i=d; // OK
        // K k=i; // NON COMPILA
        k = (D)i; // OK
        // J j=ref; // NON COMPILA
        J j=(D)ref; // OK
    }
    private static void f2() {
        K k = new D(); // OK
        I i=(C)k; // OK
    }
    private static void f3() {
        I i = new C(); // OK
        K k=(D)i; // COMPILA, ClassCastException a run-time
    }
    private static void f4() {
        L l = new E(); // OK
        K k1 = (D)l; // COMPILA, ClassCastException a run-time
        E e = new E();
        // K k2 = (D)e; // NON COMPILA, incovertible types
    }
    public static void main(String[] a) { f1(); f2(); f3(); f4(); }
}
```



Cast di reference di interfacce

Quindi, per qualsiasi reference ref che abbia come tipo statico una interfaccia I=TS(ref) è **sempre possibile convertire esplicitamente ref**, tramite l'operatore di cast, a qualsiasi tipo classe C, cioè (C)ref è sempre legale, anche quando I non è in alcuna relazione con C. Il compilatore accetta sempre questi cast.

Perchè? Questo accade perchè potrei estendere la mia gerarchia con una classe E che estende C ed implementa I, e quindi il tipo dinamico di I potrebbe essere E, sottotipo di C, e quindi il cast in questo caso andrebbe a buon fine.

La regola quindi rimane in effetti la stessa anche con le interfacce: se (C)ref potrebbe avere successo a run-time allora viene compilato correttamente.

Classi interne

Le *inner classes* (*classi interne*) permettono di annidare una definizione di classe all'interno di un'altra classe. Tipicamente, ciò viene fatto per raggruppare classi logicamente correlate e permette di controllare la visibilità dell'una dentro l'altra. Le classi interne Java sono uno strumento più sofisticato delle classi annidate del C++.

Una classe interna è considerata un **membro ordinario della classe** che la racchiude, proprio come un campo dati o un metodo, e può essere marcata con un qualsiasi livello di accesso. Una inner class ha **accesso diretto a tutti i membri di istanza**, anche quelli private, della sua classe esterna. Ciò avviene perchè un oggetto di una inner class possiede un reference nascosto all'oggetto della classe esterna che lo ho creato, una sorta di **outerThis**. Quando nella classe interna ci si riferisce ad un membro di istanza della classe esterna viene utilizzato il riferimento outerThis per selezionare quel membro. Ci si riferisce direttamente a tali membri, esattamente come fanno i membri di istanza della classe esterna. In questo rispetto, una inner class è **molto diversa dalle classi annidate** del C++.

Una **classe interna** (di istanza) **non può avere membri statici** (campi dati o metodi), perchè questa classe interna è non statica. Vedremo che sarà possibile definre anche classi interne statiche.

Classi interne

Sia Inner una classe interna ad una classe E. Dalla classe contenitrice E, l'accesso ai membri - campi dati e metodi, che sono tutti di istanza - di Inner avviene mediante oggetti di Inner.

Un **metodo di istanza** m() della classe esterna E ha il riferimento this all'oggetto di invocazione. Nel corpo di m() è possibile direttamente costruire un **oggetto innObj di Inner detenuto da this** e quindi accedere ai membri di istanza di innObj.

Invece **in un metodo statico** m() della classe esterna E non è disponibile il riferimento this e quindi **non è possibile costruire un oggetto di Inner**. Dentro m() sarà pertanto necessario avere a disposizione un oggetto ref della classe E per poter costruire un oggetto innObj di Inner detenuto da ref e successivamente accedere ai membri di innObj.

Classi interne

Consideriamo ora una **classe C esterna ad E** che può accedere ad Inner, ad esempio perchè è public. In C **è possibile dichiarare**, usando come sintassi per il tipo E.Inner, ma **non è possibile costruire direttamente** un reference della inner class Inner: bisogna necessariamente passare attraverso un oggetto di E che detenga l'oggetto di Inner. Tipicamente una classe esterna come C include un **metodo che restituisce un riferimento ad una classe interna**.

La compilazione di una classe **`E`** che contiene una classe interna **`I`** produce due file separati di bytecode: **`E.class`** e **`E$I.class`**

```
class Outer {
    int x = 1;
    public class Inner {
        private int y = 3; // campo dati y locale a Inner
        public void displayOuter() { System.out.print(" x = " + x); }
        // public static void staticMethod() {} // ILLEGALE
    } // end Inner
    public void showInner() {
        // System.out.println(y); // ILLEGALE: y è un campo dati di Inner
        System.out.print(" y = " + (new Inner()).y);
    }
    public void test() {
        Inner i = new Inner(); i.displayOuter(); showInner();
    }
    public static void n() { Outer r = new Outer();
                             Inner t = r.new Inner(); (t.y)++; }
}
public class Test {
    public static void main(String args[]) {
      Outer o = new Outer(); o.test();
      Outer.Inner i = o.new Inner(); i.displayOuter();
    }
}
// stampa: x = 1 y = 3 x = 1
```

```
public class E {
  public class Interna1 {
    private int i = 5;
    public int get1() { return i; }
  }
  public class Interna2 {
    private String s;
    Interna2(String z) { s = z; }
    String get2() { return s; }
  }
  public Interna1 i1() {
    return new Interna1();
  }
  public Interna2 i2(String z) {
    return new Interna2(z);
  }
  public void m(String z) {
    Interna1 p = i1();
    Interna2 q = i2(z);
    System.out.print(p.get1() + " "); System.out.println(q.get2());
  }
  public static void main(String[] args) {
    E e1 = new E(), e2 = new E(); e1.m("Pippo"); // stampa: 5 Pippo
    E.Interna1 x = e2.i1();
    E.Interna2 y = e2.i2("Pluto");
  }
}
```

Tipico esempio d'uso

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class MyFrame extends JFrame {
    JButton myButton; JTextArea myTextArea;
    private class ButtonListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        myTextArea.setText("bottone premuto");
      }
    }
    public MyFrame() {
      super("Inner Class Frame");
      myButton = new JButton("Premi");
      myTextArea = new JTextArea();
      add(myButton, BorderLayout.CENTER);
      add(myTextArea, BorderLayout.NORTH);
      ButtonListener bList = new ButtonListener();
      myButton.addActionListener(bList);
    }
    public static void main(String args[]) {
      MyFrame f = new MyFrame();
      f.setSize(300,300); f.setVisible(true);
    }
}
```

Classi interne

L'**occultamento** che si ottiene tramite una classe interna privata può comunque sostanzialmente essere ottenuto definendo in modo ordinario quella classe e marcandola con accesso a livello di package. L'utilità effettiva delle classi interne viene a galla quando una **classe interna implementa un'interfaccia** o deriva da una classe base. Consideriamo il seguente esempio.

```
public interface I1 { int get1(); }
public interface I2 { String get2(); }

public class E {
  private class Interna1 implements I1 { // INTERNA PRIVATA
    private int i = 5;
    public int get1() { return i; }
  }
  protected class Interna2 implements I2 { // INTERNA PROTETTA
    private String s;
    private Interna2(String z) { s = z; }
    public String get2() { return s; }
  }
  public I1 i1() { return new Interna1(); }
  public I2 i2(String z) { return new Interna2(z); }
}

public class Test {
  public static void main(String[] args) {
    E e = new E(); I1 ref1 = e.i1(); I2 ref2 = e.i2("Pippo");
    System.out.println(ref1.get1()); // stampa: 5
    System.out.println(ref2.get2());   // stampa: pippo
    //E.Interna1 r = e.new Interna1(); // Illegale: Interna1 è privata
    //E.Interna2 r = e.new Interna2("Pluto"); // Illegale: Interna2 è protetta
    }
}
```

Classi interne

Nel precedente esempio si è ottenuto un **occultamento totale** della classe privata Interna1 che implementa I1 e un occultamento protected della classe protetta Interna2 che implementa I2. Inoltre i particolari relativi alle implementazioni di I1 e I2 sono stati separati ed incapsulati in due entità separate.

La **classe E** può essere vista come "un'implementazione" delle interfacce I1 e I2 che **evita completamente qualsiasi dipendenza dalla codifica del tipo** (ad esempio i campi dati che vengono usati) e **nasconde ed incapsula completamente** i particolari relativi all'implementazione di I1 (e con accesso protected quelli di I2).

Naturalmente E non è un sottotipo di I1 e I2. Invece l'interfaccia pubblica di E include due metodi pubblici i1() e i2() che dato un oggetto di E ritornano un riferimento a I1 e I2 che si referisce ad un oggetto di una classe interna. In qualche modo**, i1() e i2()** possono essere visti come dei **metodi di "upcasting"** da E a I1 e I2. Le funzionalità previste dalle interfacce I1 e I2 si ottengono invocando queste funzionalità sui riferimenti restituiti dai metodi i1() e i2().

Il nome della classe Interna1 non è accessibile all'esterno, è un nome del tutto sconosciuto all'esterno, ed è quindi **impossibile fare downcast verso Interna1**.

Classi interne: ulteriori proprietà

Le classi interne Java sono alquanto complesse e permettono degli utilizzi piuttosto sofisticati.

Una **classe interna** può essere definita **dentro ad un qualsiasi blocco di un metodo**: tipicamente questo può venir fatto perchè il metodo deve risolvere un problema per cui l'uso di una classe potrebbe facilitare la soluzione.

Le **classi interne** possono anche essere **anonime**, cioè senza nome. Naturalmente, non avendo un nome, le **classi interne anonime non possono avere costruttori**: sarà disponibile solamente il costruttore di default standard.

Esempio

```
public interface I {
  int value();
}
public class E {
    public I i() {
        return new I() { // classe interna anonima
                private int i = 5;
                public int value() { return i; }
            }; // notare il ";"
    }
    public static void main(String[] args) {
        E e = new E(); I ref = e.i(); ref.value();
    }
}
```

Il metodo i() di E combina la costruzione e restituzione di un oggetto con la definizione della classe di cui quell'oggetto è istanza. E questa **classe interna è anonima**.

L'espressione "return new I() { classe interna };" significa: costruisco un oggetto di una classe anonima che implementa l'interfaccia I. La new in effetti ritorna un oggetto della classe anonima che viene upcastato ad I. Quindi quella sintassi di una classe interna anonima è un'abbreviazione di:

```
class Anonima implements I {
    private int i=5;  public int value() {return i;}
}
return new Anonima();
```

Classi interne statiche

Una **classe interna** può essere **statica**. Mentre un oggetto di una classe interna di istanza (non statica) ha implicitamente il riferimento outerThis all'oggetto della classe contenitrice che l'ha costruito, ciò non accade per una classe interna statica.

Quindi:

(1) **non è necessario** disporre di un oggetto della classe contenitrice per costruire un oggetto di una classe interna statica;

(2) **non esiste il riferimento outerThis** all'oggetto di invocazione della classe contenitrice, e quindi **non è possibile accedere ai membri di istanza** della classe contenitrice.

Le classi interne statiche **possono avere anche membri statici**. Se una classe interna non necessita di accedere ai membri di istanza della classe contenitrice allora la prassi consiste nel definirla statica. Le classi interne statiche sono **simili alle classi annidate del C++**.

Classi interne statiche

```
public interface I { int value(); }
public class E {
    private static class Interna implements I { // PRIVATA e STATICA
          private int i = 5;
          public int value() { return i; }
          public static int f() { return s; } // metodo statico
          private static int s = 0; // campo dati statico
    }
    public static I i() { return new Interna(); }
    public static int m() { return Interna.f(); }
}
public class Test {
    public static void main(String args[]) {
          I ref = E.i(); ref.value(); // stampa: 5
    }
}
```

Perchè le classi interne?

Generalmente una classe interna è di istanza (non statica), eredita da una classe o implementa un'interfaccia ed il codice dei metodi della classe interna manipola l'oggetto di invocazione (l'outerThis) della classe contenitrice.

Domanda: se mi serve solamente un riferimento ad un'interfaccia I perchè semplicemente non definisco una classe E che implementa I?

Risposta: se questo è quello che serve allora tipicamente si fa proprio così.

Tuttavia le classi interne hanno un'altra motivazione importante. Le classi interne possono essere considerate il completamento del supporto alla "ereditarietà multipla" in Java. Le interfacce sono una soluzione parziale, le **classi interne permettono di ereditare in modo effettivo da più di una classe** (non interfaccia). Quando si hanno a disposizione due o più classi (astratte o concrete) ed il contesto richiede di derivare da tali classi, le classi interne offrono una soluzione al problema.

Si consideri infatti il seguente esempio paradigmatico.

Perchè le classi interne?

```
public class C { public void m() {System.out.println("C.m()");} }
public abstract class A {  public abstract void m(); }

class Z extends C { // Z estende C ed internalizza A
    public void m() { System.out.println("Z.m()"); }
    A ritornaA() {
        return new A() // classe interna anonima che estende A
            { public void m() {System.out.println("Anonima.m()");} };
    }
}
public class Test {
  public static void f(C c) { c.m(); }
  public static void g(A a) { a.m(); }
  public static void main(String args[]) {
    Z z = new Z();
    f(z); // poichè Z estende C, stampa: Z.m()
    g(z.ritornaA()); // poichè Z ha una classe interna che estende A
                     // stampa: Anonima.m()
    }
}
```

Classi wrapper

In Java le variabili di tipo primitivo sono ben diverse da un reference. Esistono tuttavia le cosiddette ***classi wrapper*** (classi "involucro") per manipolare valori primitivi come fossero oggetti. L'idea è che nelle classi wrapper i valori primitivi sono "avvolti" in un oggetto "involucro" creato attorno a loro. Possono essere viste come "smart primitive types". Ogni tipo di dato primitivo ha una corrispondente classe wrapper nel package java.lang (importato per default). Ogni oggetto di una classe wrapper avvolge un singolo valore primitivo.

`int` ⇒ `Integer`; `char` ⇒ `Character`; `boolean` ⇒ `Boolean`; `byte` ⇒ `Byte`; `double` ⇒ `Double`; etc.

In ogni classe wrapper W che avvolge il tipo primitivo T è a disposizione un costruttore W(T x) che costruisce un oggetto di W avvolto al valore primitivo x di tipo T. Ci sono anche altri costruttori (vedere documentazione API)

```
int i = 27; Integer wInt = new Integer(i);
```

Classi wrapper

Le classi wrapper hanno due funzioni principali

1) Mettere a disposizione vari metodi utili che implementano **funzionalità non disponibili per il tipo primitivo**
2) Permettere di creare oggetti che memorizzano valori primitivi, affinchè possano essere manipolati da metodi che gestiscono **riferimenti ad** `Object`

Esempi

```
int stringToInt(String s) {
    return (Integer.valueOf(s)).intValue();
}

double fastFloatCalc(float x) { ... }
double slowDoubleCalc(double x) { ... }

double Calc(double x) {
    if (x >= Float.MIN_VALUE && x <= Float.MAX_VALUE)
       return fastFloatCalc((float)x); // cast sicuro
    else
       return slowDoubleCalc(x);
}
```

Classi wrapper

Per le specifiche funzionalità offerte da una classe wrapper ci si riferisce alla documentazione API. In tutte le classi wrapper sono disponibili i seguenti metodi e costruttori:

1) Un costruttore 1-ario con parametro del corrispondente tipo primitivo
2) Un costruttore 1-ario con parametro di tipo **`String`**, che, quando possibile, viene decodificato per costruire un oggetto wrapper
3) Il metodo ridefinito **`String toString()`** che ritorna una stringa che rappresenta il valore dell'oggetto di invocazione
4) Un metodo ***`tipoPrimitivo tipoPrimitivo`*`Value()`** che ritorna il valore del tipo primitivo corrispondente all'oggetto di invocazione: **`intValue()`**, **`booleanValue()`**, etc.
5) Il metodo ridefinito **`boolean equals(Object o)`** per il confronto di oggetti: questa ridefinizione confronta i valori primitivi contenuti negli oggetti e non i reference

Eccezioni ed errori

In Java **le eccezioni sono oggetti**. La classe **`java.lang.Exception`** (che estende la classe **`java.lang.Throwable`**) e tutte le sue sottoclassi definiscono un ampio insieme di condizioni di **eccezione** o **errore "non fatali"** che si possono verificare a run-time. Come in C++, il meccanismo delle eccezioni prevede del codice che gestisca l'eccezione ed eventualmente permetta di continuare l'esecuzione.

Ogni condizione anormale che influenza il normale flusso di esecuzione del programma è una ***eccezione*** o un ***errore irrecuperabile***. La classe **`java.lang.Error`** (che è pure sottoclasse di **`java.lang.Throwable`**) definisce le condizioni di errore "irrecuperabile", che non dovrebbero essere gestite. Se si verifica una tale condizione di errore, il programma dovrebbe terminare.

La gestione delle eccezioni in Java è simile al C++. Le keywords da usare sono **`try`**, **`catch`**, **`throw`**, **`throws`** e **`finally`**. Tutte e sole le eccezioni che un metodo di qualche classe dell'API può lanciare sono definite nella sua documentazione, sotto la clausola **`throws`**.

Eccezioni

Esempi di eccezioni

- Si tenta di aprire un file che non esiste (`FileNotFoundException`)
- Non si riesce a stabilire una connessione di rete (`ServerNotActiveException`)
- Divisione per zero (`ArithmeticException`)
- Si tenta di caricare una classe che non esiste o che non si riesce a trovare (`ClassNotFoundException`)

```
public class C {
    public static void main (String args[]) {
        int i = 0; String[] s = {"Pippo", "pippo", "PIPPO"};
        while (i < 4) { System.out.print(s[i]); i++; }
    }
}
/* STAMPA:
PippopippoPIPPO
E POI:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
*/
```

```
public class Z { public void m() {} }
public class C {
    private Z z;
    public void n() { z.m(); }
    public static void main (String args[]) {
        C ref = new C(); ref.n();
    }
}
/* A RUN-TIME:
Exception in thread "main" java.lang.NullPointerException
  at C.n(C.java:6)
  at C.main(C.java:8)
*/
```

try e catch

Il meccanismo try/catch è identico al C++. Il codice di un metodo m() che in esecuzione potrebbe lanciare un'eccezione può venire incluso in un blocco try, seguito da una lista di blocchi catch adiacenti. Un blocco try viene eseguito fino al lancio di un'eccezione o fino alla sua normale terminazione senza alcun lancio di eccezioni. Se viene sollevata un'eccezione di tipo E, vengono **esaminate in ordine le catch** per trovare un eventuale **gestore** per un'eccezione di quella classe E o, per subtyping, **di una qualsiasi superclasse** di E. Se questa ricerca trova un tale blocco catch per l'eccezione di tipo E, si esegue il corrispondente codice della catch, e nessuna altra catch che segue verrà esaminata. Se non si trova un blocco catch(T), con $E \leq T$, l'eccezione viene rilanciata al metodo che ha invocato il metodo m(), ed un eventuale try esterno potrebbe gestirla. Al termine dell'esecuzione di un blocco catch che gestisce un'eccezione, il controllo ritorna al codice che immediatamente segue la lista dei blocchi catch. Vi può essere un numero qualsiasi di catch, anche nessuna, purchè catturino **tipi diversi** di eccezioni. Quindi, il polimorfismo (ma non il linguaggio o il compilatore) **ragionevolmente impone** di definire le catch nell'ordine del tipo catturato da sottotipo a supertipo. Come vedremo nel seguito, alcuni tipi di eccezioni devono obbligatoriamente essere gestite o con un try/catch o con una throws.

Rilancio di eccezioni

Quindi se un qualsiasi statement S (tipicamente un'invocazione di metodo) nel corpo di un metodo m() provoca un'eccezione che non viene catturata da un blocco try/catch (possibilmente perchè lo statement S non era incluso in un blocco try), questa eccezione viene **rilanciata al metodo** che ha invocato m(). Questo processo si ripete seguendo il call stack dei metodi, fintantochè l'eccezione non viene catturata da una catch. Se l'eccezione arriva al metodo main() e nemmeno nel main() viene catturata, l'eccezione provoca la terminazione abnormale del programma (e la visualizzazione sulla shell delle informazioni sull'eccezione).

finally

La keyword **`finally`** permette di definire un blocco di codice che verrà **sempre e comunque eseguito**, qualsiasi eccezione a run-time possa essere sollevata ed eventualmente catturata. La motivazione è la stessa della catch generica del C++, ma il funzionamento è diverso perchè il **codice all'interno della `finally` viene sempre e comunque eseguito**.

Esempio "intuitivo" classico:

```
try {
        apriRubinetto(); riempiVasca();
} finally {
        chiudiRubinetto();

}
```

Anche se viene sollevata una qualche eccezione, si deve comunque chiudere il rubinetto. Tipico esempio: l'apertura e seguente lettura da un file può sollevare delle eccezioni, ma il file dovrà comunque essere chiuso per rilasciare la disponibilità della risorsa condivisa con il resto del codice.

```
public class Loop {
    public static void main (String args[]) {
        int i = 0; String[] s = {"Pippo", "PIPPO", "pippo"};
        while (i < 4) {
            try {
                System.out.print(s[i] + " ");
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Reset dell'indice");
                i = -1;
            } finally {
                System.out.println("Questo lo stampo sempre");
            }
            i++;
        }
    }
}
/* Vanno in loop le seguenti stampe:
Pippo Questo lo stampo sempre
PIPPO Questo lo stampo sempre
pippo Questo lo stampo sempre
Reset dell'indice
Questo lo stampo sempre
*/
```

Tipi di eccezioni

La classe **`java.lang.Throwable`** è superclasse di ogni oggetto eccezione ed errore che può essere sollevato e catturato. I metodi di **`Throwable`** (vedi API) gestiscono le stringhe d'errore da stampare e tracciano lo stack delle chiamate. Non è buona norma usare la classe generica **`Throwable`**. Ci dono tre sottotipi importanti di **`Throwable`**:

1) **`Error`**: è sottoclasse diretta di **`Throwable`** ed indica severi (o fatali) **errori difficili**, se non impossibili, **da gestire**, come "out of memory" o "JVM error". Usualmente un programma non gestisce tali condizioni. Esempi di sottoclassi: **`IOError`**, **`VirtualMachineError`**
2) **`RuntimeException`**: è una sottoclasse diretta di **`Exception`**, ed indica un **problema progettuale logico o di implementazione**, che non dovrebbe verificarsi qualora il programma fosse "logicamente corretto". Esempi di sottotipi: **`ArithmeticException`**, **`ArrayIndexOutOfBoundsException`**, **`NullPointerException`**. La prassi è pertanto quella di **non gestire queste eccezioni**, lasciando che il programma interrompa l'esecuzione con un messaggio a run-time.
3) Tutte le altre eccezioni, quindi tutte le eccezioni il cui tipo E è tale che E≤**`Throwable`**, **not**(E≤**`Error`**) e **not**(E≤**`RuntimeException`**), **devono obbligatoriamente essere gestite**, altrimenti il compilatore segnala una illegalità. Esempio tipico: tutte i sottotipi di **`java.io.IOException`**.

Le eccezioni in 1) e 2) sono dette ***non controllate***, quelle in 3) ***controllate***

`RuntimeException`s comuni

- **`ArithmeticException`**: tipicamente una divisione per zero
- **`NullPointerException`**: un tentativo di accedere un campo dati o di invocare un metodo con un reference che non si riferisce ad un oggetto effettivo, ovvero per un reference null. **Attenzione**: è probabilmente l'errore logico più comune!

```
Date d = null;
System.out.print(d.toString());
```

- **`ClassCastException`**: lanciata quando un cast dinamico non è andato a buon fine
- **`ArrayIndexOutOfBoundsException`**: un tentativo di accedere un elemento di un'array oltre i bounds

Gestire le eccezioni controllate

Java richiede che **ogni metodo m() che contiene una invocazione di un metodo e() che può sollevare una eccezione controllata deve obbligatoriamente prevedere di gestire quell'eccezione**. Altrimenti, il compilatore segnala una illegalità. Ci sono due possibilità per il metodo m() di adempiere all'obbligo di gestire una eccezione:

1) Il metodo m() gestisce una eccezione di tipo E che può essere sollevata dalla chiamata di e() **inserendo la chiamata ad e() in un blocco try/catch**, dove il tipo della catch è o E oppure un supertipo di E. È possibile che il codice della catch sia vuoto, ossia avere una **gestione fittizia,** nonostante ciò sia una **bad practice**, quindi sconsigliata
2) Il metodo m() **dichiara che non gestirà l'eccezione di tipo E** che può sollevare la chiamata di e(), e di conseguenza l'eccezione sarà rilanciata al metodo chiamante di m(). Ciò si ottiene usando la keyword `throws`:

```
T m(T1 x1,...,Tn xn) throws Exception1, Exception2, ...
```

Anche il metodo main() deve obbligatoriamente gestire le eccezioni controllate. La scelta tra le due possibilità (1) e (2) valuterà l'opportunità di considerare il metodo chiamante m() o qualche chiamante di m() come il candidato più appropriato per gestire l'eccezione.

Le eccezioni sono organizzate in una gerarchia di classi (vedi API), e quindi spesso si **gestiscono degli opportuni supertipi di eccezioni** con lo stesso codice: esempio tipico è il supertipo `IOException`

Eccezioni user-defined

Come in C++, è possibile **definire delle nuove eccezioni** mediante una classe che **estende `Exception`**. Queste sono classi "ordinarie" a tutti gli effetti, e quindi possono contenere campi dati, costruttori e metodi. La keyword **`throw`** permette di sollevare una eccezione di qualsiasi tipo, in particolare user-defined. Quindi se **`exc`** è un reference di un tipo **`E`** che è sottotipo di **`Exception`**, con lo statement **`throw exc;`** il controllo lascia il metodo contenente questo statement e si trasferisce alla prima catch che nel call stack cattura l'eccezione, oppure se l'eccezione arriva al metodo main() senza essere catturata provoca la terminazione del programma.

```
throw new EOFException("EOF nella lettura");
throw new IllegalArgumentException();
throw new MyException("Eccomi qui!");
```

```
public class ServerTimeOutException extends Exception {
    private int port;
    public ServerTimeOutException(String reason, int port) {
          super(reason); // chiamata al costruttore di Exception
          this.port = port;
    }
    // Throwable.getMessage()
    public String getReason() { return this.getMessage(); }
    public int getPort() { return port; }
}
```

```
public class ServerTimeOutException extends Exception {
    private int port;
    public ServerTimeOutException(String reason, int port) {
          super(reason); // chiamata al costruttore di Exception
          this.port = port;
    }
    // Throwable.getMessage()
    public String getReason() { return this.getMessage(); }
    public int getPort() { return port; }
}
```

```
public class Connessione {
  public static int open(String s, int porta) { ... }
  public static void connect(String serverName) throws ServerTimeOutException {
    int porta = 80, successo = open(serverName, porta);
    if (successo == -1) throw new ServerTimeOutException("Could not connect", porta);
    }
  public static void findServer() {
    try { connect(defaultServer); }
    catch (ServerTimeOutException e) {
      System.out.println("Server time-out: try another server");
      try { connect(alternativeServer); }
      catch (ServerTimeOutException e1) {
         System.out.print("Error:"+e1.getReason()+" connecting to port "+e1.getPort());
      }
    }
  }
}
```

Classi generiche

Java 5 ha introdotto i "generics", cioè **classi generiche e metodi generici parametrizzati sui tipi**, ossia il supporto Java alla programmazione generica (in C++ sono i template). Ad esempio, la classe **`Vector`** della libreria è generica, cioè è stata dichiarata usando un parametro di tipo **`E`**. Per istanziare una classe generica si devono fornire dei tipi effettivi per i parametri di tipo.

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable {
  public boolean add(E obj) {...}
  public E elementAt(int index) {...}
  ...
}

Vector<String> v = new Vector<String>();
Vector<Dipendente> w = new Vector<Dipendente>(5);
v.add(new String("pippo"));
String s = v.elementAt(0);
// w.add(new String("pippo")); // NON COMPILA
```

Classi generiche

I parametri di tipo di un generics possono essere **istanziati a classi ed interfacce ma non a tipi primitivi**.

Prima dei generics, la classe **`Vector`** sfruttava **`Object`**:

```
Vector v = new Vector();
v.add(new String("pippo")); v.add(new Integer(1));
String s = (String)v.elementAt(0);
Integer i = (Integer)v.elementAt(1);
```

Classi generiche

Esempio di una semplice classe generica **`Coppia<T,S>`**

```
public class Coppia<T, S> {
  private T first;
  private S second;
  public Coppia(T x,S y) {first = x; second = y;}
  public T getFirst() {return first;}
  public S getSecond() {return second;}
}
```

Metodi generici

Un **metodo generico** è un metodo che ha una o più variabili di tipo. Quando si invoca un metodo generico non si devono specificare i tipi effettivi da usare al posto dei parametri di tipo: si invoca semplicemente il metodo con i parametri appropriati e il compilatore dedurrà i tipi da usare per i parametri.

Esempio: metodo statico che stampa tutti gli elementi di un array

```
public class C {
  public static <E> void print(E[] a) {
    for(int i=0; i<a.length; ++i) System.out.print(a[i]);
    System.out.println();
  }
}

Dipendente[] a1 = new Dipendente[9];
Integer[] a2 = new Integer[5]
...
C.print(a1);
C.print(a2);
```

Vincolare variabili di tipo

Spesso è necessario specificare quali tipi possano essere usati in una classe generica oppure in un metodo generico. Ad esempio, in un metodo generico **`E min(E[] a)`** che trova il valore minimo presente in un array **`a`** è necessario specificare che il parametro di tipo **`E`** implementi l'interfaccia **`Comparable`**. Dobbiamo quindi **vincolare il parametro di tipo `E`** nel seguente modo:

```
public static <E extends Comparable> E min(E[] a) {
  E x = a[0];
  for(int i=1; i<a.length; ++i) if(a[i].compareTo(x) < 0) x=a[i];
    return x;
}
```

Vincolare variabili di tipo

```
public static <E extends Comparable> E min(E[] a) {
  E x = a[0];
  for(int i=1; i<a.length; ++i) if(a[i].compareTo(x) < 0) x=a[i];
    return x;
}
```

È possibile invocare **`min`** con un array di **`String`**, che implementa Comparable, ma non con un array di **`JButton`**. Se non ci fosse il vincolo per **`E`** di essere un sottotipo di **`Comparable`** il metodo **`min`** non sarebbe stato compilato.

Per esprimere il vincolo che **`E`** debba essere un sottotipo di due tipi **`T1`** e **`T2`** si usa la sintassi **`<E extends T1 & T2>`**

Generics e subtyping

`Dirigente` è un sottotipo di **`Dipendente`**. È vero che **`Vector<Dirigente>`** è un sottotipo di **`Vector<Dipendente>`**?

NO! Una relazione di subtyping tra i parametri di tipo non induce una relazione di subtyping tra le corrispondenti istanziazioni di una classe generica.

```
class Dirigente extends Dipendente {...}
class Agente extends Dipendente {...}

Vector<Dirigente> a = new Vector<Dirigente>();
// se fosse possibile...
Vector<Dipendente> b = a;
Dipendente dip = new Agente();
b.add(dip); // ...non andrebbe bene!!
```

Generics e subtyping

```
class Dirigente extends Dipendente {...}
class Agente extends Dipendente {...}

Vector<Dirigente> a = new Vector<Dirigente>();
// se fosse possibile...
Vector<Dipendente> b = a;
Dipendente dip = new Agente();
b.add(dip); // ...non andrebbe bene!
```

Attenzione: con gli **array invece non c'è alcun controllo**

```
class B {}  class C extends B {}   class D extends B {}

B[] ab = new B[3];
C[] ac = new C[5];
ab = ac;   // OK, tipa perché C[] è sottotipo di B[]
B ref = new D();
ab[0]=ref; // COMPILA CORRETTAMENTE
// a run-time: java.lang.ArrayStoreException
```

Wildcards

Nell'istanziazione di una classe generica si possono specificare vincoli per i parametri di tipo usando **wildcards** (caratteri jolly).

Nome	Sintassi	Significato
vincolo con limite inferiore	`? extends B`	qualsiasi sottotipo di `B`
vincolo con limite superiore	`? super B`	qualsiasi supertipo di `B`
nessun vincolo	`?`	qualsiasi tipo

```
class Vector<E> ... {
public boolean addAll(Collection<? extends E> c);
public boolean containsAll(Collection<?> c);
public void sort(Comparator<? super E> c);
}
```

Wildcards

Nome	Sintassi	Significato
vincolo con limite inferiore	`? extends B`	qualsiasi sottotipo di `B`
vincolo con limite superiore	`? super B`	qualsiasi supertipo di `B`
nessun vincolo	`?`	qualsiasi tipo

```
interface java.lang.Comparable<T> {...}
public static <E extends Comparable<? super E> > E min(E[] a) {...}

// Il seguente vincolo sarebbe troppo restrittivo
public static <E extends Comparable<E> > E min2(E[] a)
// ESEMPIO
Dipendente implements Comparable<Dipendente> ...
Dirigente extends Dipendente ...
// Ma Dirigente non è sottotipo di Comparable<Dirigente> quindi
min2(new Dirigente[4]); // NON COMPILA
// mentre
min(new Dirigente[4]);  // COMPILA
```

Raw types

La JVM non funziona con riferimenti a tipi generici ma solo con **raw types** (**tipi grezzi**), ovvero i parametri di tipo di un tipo generico sono sostituiti da tipi Java ordinari. Ciascun parametro di tipo viene **sostituito dal suo tipo effettivo di istanziazione se disponibile oppure da `Object`** se il tipo effettivo non è indicato. Il compilatore rimuove ogni riferimento ai parametri di tipo nel momento in cui compila classi/metodi generici.

La classe **`Coppia<T,S>`** viene compilata nel seguente raw type:

```
public class Coppia {
  private Object first;
  private Object second;
  public Coppia(Object x,Object y) {
    first = x; second = y;
}
  public Object getFirst() {return first;}
  public Object getSecond() {return second;}
}
```

Restrizioni nei generics

Non è possibile costruire nuovi oggetti che siano **istanze di un parametro di tipo**.

```
public static <E> void riempiConDefaults(E[] a) {
  for(int i=0; i<a.length; ++i)
    a[i]=new E(); // NON COMPILA
}
// Infatti sarebbe compilato in
public static void riempiConDefaults(Object[] a) {
  for(int i=0; i<a.length; ++i)
    a[i]=new Object();
}
```

Restrizioni nei generics

Non è possibile usare dei parametri di tipo di una classe generica per definire campi dati **statici**, metodi **statici** o classi interne **statiche**.

```
public class List<E> {
  private static E defaultValue; // ILLEGALE
  public static List<E> replicate(E value, int n) {...} // ILLEGALE
  private static class Nodo {
    public E info; // ILLEGALE
    public Nodo next;
  }
}
```

Per i metodi statici e le classi interne statiche basta aggiungere un tipo parametrico.

```
public class List<E> {
public static <T> List<T> replicate(T value, int n) {...} // OK
  private static class Nodo<T> {
    public T info; // OK
    public Nodo<T> next;
  }
}
```