

OOP IN JAVA

Introduzione a Java

Java è un linguaggio di programmazione nato all'inizio degli anni novanta da un gruppo di lavoro della **Sun Microsystems** guidato da **James Gosling**

Nasce nel periodo in cui i linguaggi più utilizzati erano **C/C++**

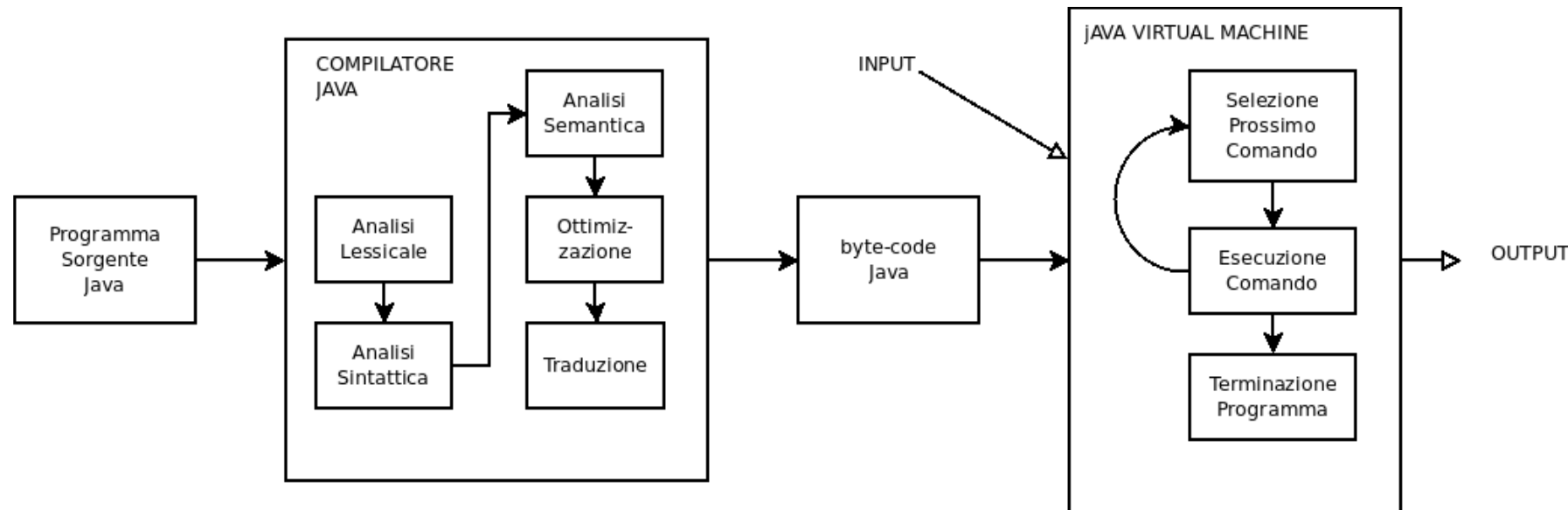
Obiettivi nello sviluppo del linguaggio Java:

- ▶ **Semplificare la programmazione** rispetto a C/C++ (es. garbage collection per la gestione della memoria allocata)
- ▶ **Indipendenza dalla piattaforma** (lo stesso codice compilato eseguibile su qualunque hardware supportato, senza ricompilare)
- ▶ **Modularizzabilità** per favorire lo sviluppo collaborativo (OOP)
- ▶ **Robustezza e sicurezza** con controlli statici e dinamici per prevenire e controllare comportamenti anomali dei programmi

Approccio: compilazione + interpretazione

Si è scelto un approccio di **compilazione + interpretazione**, con un **bytecode** intermedio

- ▶ Il **bytecode** è utilizzabile in **ogni architettura** per cui sia disponibile un interprete (Java Virtual Machine), senza ricompilare
- ▶ Consente controlli **statici** (compilatore) + **dinamici** (interprete)



Uso di Java

Fine anni '90 - Applet Java: programmazione di componenti interattive delle pagine web (es. videogiochi, interfacce interattive)

- ▶ **JVM come plugin** del browser e bytecode inviato dal sito ed interpretato localmente (**security issues!**)

Anni '00 in poi - Applicazioni desktop: grosse applicazioni sviluppate in team ed eseguibili su S.O. diversi

- ▶ Particolarmente adatto per lo sviluppo di **applicazioni distribuite** che comunicano sulla rete

Anni '10 in poi - App Android: Java è il linguaggio di riferimento per lo sviluppo di app native in ambiente Android (**security issues!**)

Ambiente di sviluppo (JDK e IDE)

Java Development Kit (JDK) - <http://jdk.java.net/>

Compilatore + Interprete (JVM) + API + tool di supporto allo sviluppo

IDE ed editor principali:

- ▶ Eclipse: <https://www.eclipse.org/ide/>
- ▶ IntelliJ IDEA: <https://www.jetbrains.com/idea/>
- ▶ NetBeans: <https://netbeans.apache.org/>
- ▶ VSCode: <https://code.visualstudio.com/>

Struttura di un programma Java

Un programma java è un **insieme di classi**

- ▶ In linea di principio, **ogni classe in un file diverso** (con lo stesso nome della classe ed estensione `.java`)
- ▶ Una classe dovrà avere un **metodo `main`** da cui parte l'esecuzione del programma

```
public class HelloWorld {  
    public static void main (String [] args) {  
        // visualizza un messaggio di saluto  
        System.out.println("Hello World !");  
    }  
}
```

Per eseguire (da terminale):

1. Salvare come `HelloWorld.java`
2. Compilare con `javac HelloWorld.java`
(si ottiene il bytecode `HelloWorld.class`)
4. Eseguire con `java HelloWorld`

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

TIPI PRIMITIVI

Nome del tipo	Rappresentazione	Memoria usata
byte	Interi in complemento a 2	1 byte
short	Interi in complemento a 2	2 byte
int	Interi in complemento a 2	4 byte
long	Interi in complemento a 2	8 byte
float	Virgola mobile (singola precisione)	4 byte
double	Virgola mobile (singola precisione)	8 byte
char	Codifica Unicode	2 byte
boolean	Valore di verità (true/false)	1 bit*

* In realtà le implementazioni note della JVM di solito usano **1 byte** per questioni di indirizzamento

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

TIPI PRIMITIVI

Nome del tipo	Rappresentazione	Memoria usata
byte	Interi in complemento a 2	1 byte
short	Interi in complemento a 2	2 byte
int	Interi in complemento a 2	4 byte
long	Interi in complemento a 2	8 byte
float	Reali in formato IEEE 754	4 byte
double	Reali in formato IEEE 754	8 byte
char	Caratteri Unicode	2 byte
boolean	Valore di verità (true/false)	1 bit*

ATTENZIONE: Esiste un tipo **String** per rappresentare le stringhe. Non è un tipo primitivo (String è una classe di libreria) ma è gestito in modo speciale.

Le stringhe hanno tipo String e si possono concatenare con + come se String fosse un tipo primitivo

* In realtà le implementazioni note della JVM di solito usano **1 byte** per questioni di indirizzamento

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

ESPRESSIONI: come in C, ma con in aggiunta `true` e `false`

Esempi: `(3+4.0)*7/2.0` `(x>0) && (x<=100)` `(x==0) || (x!=0) || true`

COMANDI: assegnamenti, blocchi, condizionali e cicli come in C

Esempi: `x=6` `if (x>0) {...} else {...}` `while (x>0) {...}`
`do {...} while (x>0)` `for (int i=0; i<10; i++) {...}`
`return 10`

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

DICHIARAZIONI DI VARIABILI:

- ▶ Le **variabili locali** devono essere **dichiarate** e **inizializzate** prima di poterle usare (**controllo statico**)
- ▶ **Non è possibile ridichiarare** in un blocco una variabile già dichiarata in un blocco più esterno (**no shadowing**)
- ▶ **Non esistono variabili globali** (vedremo che si possono definire variabili visibili a livello di classe, ma non di intero programma)

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

DICHIARAZIONI DI VARIABILI (ESEMPIO):

```
... // nel corpo di un metodo
int x;
int y=0;
int z;
if (x>0) {      // ERRORE: x non è inizializzata
    z=10;
    int k=z;    // OK: z inizializzata un attimo prima
    char y='c'; // ERRORE: non si può ridichiarare y
    ...
}
...
```

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

ARRAY (simili a C ma non troppo):

```
// dichiarazione
int[] numeri;    // attenzione alle parentesi:
                  // «int numeri[]» è sbagliato

// inizializzazione
numeri = new int[10]; // dimensione 10 e valori di default (0)
numeri = {5,3,2,6};   // dimensione 4 e valori 5,3,2 e 6

// anche insieme
int[] numeri = new int[10];
```

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

ARRAY (ESEMPI):

```
public void popola(int[] numeri) {  
    for (int i=0; i<numeri.length; i++) {  
        numeri[i]=i;  
    }  
}  
  
public int somma(int[] numeri) {  
    int somma = 0;  
    for (int i=0; i<numeri.length; i++) {  
        somma+=numeri[i];  
    }  
    return somma;  
}
```

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

ARRAY (ESEMPI):

```
public void popola(int[] numeri) {  
    for (int i=0; i<numeri.length; i++) {  
        numeri[i]=i;  
    }  
}
```

```
public int somma(int[] numeri) {  
    int somma = 0;  
    for (int i=0; i<numeri.length; i++) {  
        somma+=numeri[i];  
    }  
    return somma;  
}
```

Quando l'indice non serve, si può iterare su un array con un **for-each**:

```
public int somma(int[] numeri) {  
    int somma = 0;  
    for (int n: numeri) somma+=n;  
    return somma;  
}
```

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

PUNTATORI:

- ▶ **NON CI SONO!** (ma ci saranno riferimenti a oggetti...)
- ▶ A differenza del C, gli array possono quindi essere acceduti solo tramite l'indice, e non trattandoli come puntatori
- ▶ A **run-time**, prima di accedere ad un array, la JVM controlla che l'indice sia compreso tra 0 e length-1, altrimenti solleva una eccezione di tipo **ArrayIndexOutOfBoundsException**

Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

FUNZIONI:

- **SI USANO I METODI** (che possono essere ricorsivi, anche mutuamente)

Modificatore di
visibilità

Tipo del
risultato

Parametri
formali

```
public int fattoriale(int n) {  
    int res=1;  
    if (n==0)  
        return res;  
    else  
        return n * fattoriale(n-1);  
}
```


Nucleo imperativo

Il corpo dei metodi è **codice imperativo** con una **sintassi molto simile al C**

FUNZIONI ANONIME:

- ▶ Si possono definire funzioni anonime (**lambda-espressioni**) con la seguente sintassi:

$x \rightarrow \text{espressione}$	$(x,y,..) \rightarrow \text{espressione}$
$x \rightarrow \{ \text{blocco} \}$	$(x,y,..) \rightarrow \{ \text{blocco} \}$

- ▶ Non essendo però Java un linguaggio funzionale, il funzionamento di queste espressioni è un po' particolare
 - ▶ **Ne parleremo più avanti...**

Classi

- ▶ Una classe consiste di **variabili** e **metodi**, e può prevedere uno o più **costruttori** per inizializzare le variabili
- ▶ **PRINCIPIO DI ATRAZIONE:** *“Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts”*
- ▶ **Cioé:** ogni **tipologia di “entità”** identificabile nel programma come caratterizzata da un **comportamento autonomo** (o un **insieme di comportamenti correlati**) dovrebbe corrispondere a una diversa classe

Classi: esempio ContoCorrente

- ▶ Vediamo un **esempio** di programma che fa operazioni su un conto corrente
- ▶ Consiste di **due classi**:
 - ▶ **Banca** che modella la banca e contiene il `main` del programma
 - ▶ **ContoCorrente** che descrive gli oggetti che rappresentano conti correnti

Classi: esempio Banca

```
public class Banca {  
    public static void main(String[] args) {  
  
        // crea un conto inizializzato con 1000 euro  
        ContoCorrente conto1 = new ContoCorrente(1000);  
  
        // crea un conto inizializzato con 200 euro  
        ContoCorrente conto2 = new ContoCorrente(200);  
  
        // si assicura che ci siano 700 euro nel conto1 ...  
        if (conto1.saldo >= 700) {  
            // ... preleva 700 euro dal primo conto...  
            conto1.preleva(700);  
            // ... e li versa nel secondo  
            conto2.versa(700);  
        }  
  
        System.out.println("Saldo primo conto: " + conto1.saldo );  
        System.out.println("Saldo secondo conto: " + conto2.saldo );  
    }  
}
```

Classi: esempio ContoCorrente

```
public class ContoCorrente {  
    // variabile che rappresenta lo stato (visibile in tutta la classe)  
    public double saldo;  
  
    // costruttore (stesso nome della classe e senza tipo di ritorno)  
    public ContoCorrente(double saldoIniziale) {  
        saldo = saldoIniziale;  
    }  
  
    // metodi  
    public void versa(double somma) {  
        saldo += somma;  
    }  
    public void preleva(double somma) {  
        saldo -= somma;  
    }  
}
```

Per eseguire (da terminale):

1. Salvare le due classi come Banca.java e ContoCorrente.java
2. Compilare con `javac *.java`
(ogni classe viene **compilata separatamente**)
4. Eseguire con `java Banca`

Classi: esempio ContoCorrente

Osservazioni:

- ▶ Abbiamo usato il metodo `main` di Banca **senza creare oggetti** di questa classe
 - ▶ E' un **metodo statico** (per questo c'è il modificatore `static`) ossia può essere richiamato anche senza istanziare la classe
 - ▶ Per forza di cose, il metodo `main` è sempre statico...
- ▶ La classe Banca **può accedere** a tutti i membri di ContoCorrente (sia alle variabili che ai metodi)
 - ▶ Perché li abbiamo definiti **pubblici** (questo è il significato di `public`)
 - ▶ **Potremmo evitarlo** usando invece il modificatore `private`

Classi: esempio ContoCorrente

Arricchiamo un po' il comportamento di ContoCorrente:

- ▶ Tracciamo i movimenti stampando dei messaggi

```
public class ContoCorrente {
    public double saldo;

    public ContoCorrente(double saldoIniziale) {
        saldo = saldoIniziale;
    }

    public void versa(double somma) {
        saldo += somma;
        System.out.println("Versati: " + somma + " euro"); // aggiunto
    }
    public void preleva(double somma) {
        saldo -= somma;
        System.out.println("Prelevati: " + somma + " euro"); // aggiunto
    }
}
```

Classi: esempio ContoCorrente

Arricchiamo un po' il comportamento di ContoCorrente:

- ▶ Tracciamo i movimenti stampando dei messaggi

```
public class ContoCorrente {  
    public double saldo;  
  
    public ContoCorrente(double saldoIniziale) {  
        Domande:  
        - Dobbiamo modificare in qualche modo la classe Banca?  
        - Dobbiamo ricompilare entrambe le classi?  
        System.out.println("Versati: " + somma + " euro"); // aggiunto  
    }  
    public void preleva(double somma) {  
        saldo -= somma;  
        System.out.println("Prelevati: " + somma + " euro"); // aggiunto  
    }  
}
```


Interfaccia pubblica di una classe

- ▶ Le modifiche che abbiamo apportato a ContoCorrente **non influenzano l'interazione** tra le due classi
- ▶ La classe Banca potrà continuare a chiamare i metodi `versa` e `preleva` come faceva prima, perché ne abbiamo modificato solo il contenuto (l'implementazione)
- ▶ Se non modifichiamo l'**INTERFACCIA PUBBLICA** di una classe (nomi dei metodi e variabili pubbliche, parametri di tali metodi e tipi dei valori di ritorno) possiamo modificare e ricompilare la classe senza compromettere il resto del programma (**COMPOSIZIONALITA'**)

Ancora modifiche a ContoCorrente

Abbiamo aggiunto delle stampe in ContoCorrente per monitorare le operazioni di saldo e versamento

- ▶ Ma chi vieta all'utilizzatore del conto corrente (la Banca) di modificare a mano il saldo sfuggendo al tracciamento?

```
conto1.saldo = 100000; // modifica che non genera stampe
```

- ▶ La variabile `saldo` è **pubblica**, quindi chiunque dall'esterno la può modificare...
- ▶ **Soluzione:** specifichiamo `saldo` come privata!
(**INFORMATION HIDING**)


Ancora modifiche a ContoCorrente

```
public class ContoCorrente {  
    private double saldo; // NASCOSTA!  
  
    public ContoCorrente(double saldoIniziale) { ...  
    public void versa(double somma) { ... }  
  
    public double getSaldo() { return saldo; } // aggiunto  
  
    public boolean preleva(double somma) { // modificato  
        if (saldo >= somma) {  
            saldo -= somma;  
            System.out.println("Prelevati: " + somma + " euro");  
            return true;  
        }  
        else return false;  
    }  
}
```

Aggiungiamo `getSaldo`
per consentire accesso al
saldo in sola lettura



Aggiungiamo anche un controllo sul saldo fatto
direttamente dal metodo `preleva`!



Ancora modifiche a ContoCorrente

```
public class ContoCorrente {  
    private double saldo; // NASCOSTA!
```

```
    public ContoCorrente(double saldoIniziale) { ...  
    public void versa(double somma) { ... }
```

Aggiungiamo `getSaldo`
per consentire accesso al
saldo in sola lettura

Domande:

- Dobbiamo modificare in qualche modo la classe Banca?
- Dobbiamo ricompilare entrambe le classi?

```
    if (s  
        saldo -= somma;  
        System.out.println("Prelevati: " + somma + " euro");  
        return true;  
    }  
    else return false;  
}
```

Aggiungiamo anche un controllo sul saldo fatto
direttamente dal metodo `preleva`!

```
}
```

Ancora modifiche a ContoCorrente

Questa volta abbiamo modificato l'**INTERFACCIA PUBBLICA** della classe:

- ▶ `saldo` non è più nell'interfaccia pubblica
- ▶ `getSaldo` è stato aggiunto nell'interfaccia pubblica
- ▶ `preleva` ha un tipo di risultato diverso

In effetti, la classe `Banca` ora **non è più corretta**

- ▶ La **modifichiamo**, e **ricompiliamo** entrambe le classi...

Classe Banca (vecchia versione)

```
public class Banca {  
    public static void main(String[] args) {  
  
        // crea un conto inizializzato con 1000 euro  
        ContoCorrente conto1 = new ContoCorrente(1000);  
  
        // crea un conto inizializzato con 200 euro  
        ContoCorrente conto2 = new ContoCorrente(200);  
  
        // si assicura che ci siano 700 euro nel conto1  
        if (conto1.saldo >= 700) {  
            // ... preleva 700 euro dal primo conto  
            conto1.preleva(700);  
            // ... e li versa nel secondo  
            conto2.versa(700);  
        }  
  
        System.out.println("Saldo primo conto: " + conto1.saldo );  
        System.out.println("Saldo secondo conto: " + conto2.saldo );  
    }  
}
```

saldo non è più visibile

Preleva restituisce un
booleano che stiamo
ignorando

Classe Banca (nuova versione)

```
public class Banca {  
    public static void main(String[] args) {  
  
        // crea un conto inizializzato con 1000 euro  
        ContoCorrente conto1 = new ContoCorrente(1000);  
  
        // crea un conto inizializzato con 200 euro  
        ContoCorrente conto2 = new ContoCorrente(200);  
  
        // effettua il bonifico  
        if (conto1.preleva(700)) { // prelievo con controllo!!  
            conto2.versa(700);  
        }  
  
        // usa getSaldo  
        System.out.println("Saldo primo conto: " + conto1.getSaldo() );  
        System.out.println("Saldo secondo conto: " + conto2.getSaldo() );  
    }  
}
```

Incapsulamento

I modificatori `public` e `private` consentono di realizzare meccanismi di **INCAPSULAMENTO**

- ▶ La **rappresentazione dell'oggetto** (le variabili) rimane nascosta (**privata**)
- ▶ L'accesso dall'esterno è consentito solo tramite (e sotto il controllo di) un certo numero di **metodi pubblici**
- ▶ Dall'esterno si potrebbe **non conoscere la rappresentazione** (es. per il saldo si potrebbe «di nascosto» usare una lista per tenere traccia dei movimenti mantenendo la stessa interfaccia pubblica)
- ▶ E' possibile anche definire **metodi privati** (diventano metodi ausiliari utilizzabili solo all'interno della classe stessa)

Interfacce

L'interfaccia pubblica di una classe può essere esplicitata usando uno specifico costrutto del linguaggio Java:

- ▶ Le **INTERFACCE**

Una interfaccia in Java contiene una **specifica astratta della classe**: indica quali membri pubblici la classe deve contenere

- ▶ Rende le classi una sorta di **Abstract Data Type (ADT)**

Esempio: Interfaccia BankAccount

Specifichiamo un'interfaccia per il conto corrente:

- ▶ Contiene solo le intestazioni dei membri pubblici
- ▶ No costruttori

```
public interface BankAccount {    // interface, non class

    public double getSaldo();
    public void versa(double somma);
    public boolean preleva(double somma);

}
```

Esempio: Interfaccia BankAccount

Ora diciamo esplicitamente che ContoCorrente **implementa** l'interfaccia BankAccount:

► Nella classe aggiungiamo `implements BankAccount`

```
public class ContoCorrente implements BankAccount { // implements!
    private double saldo;
    public ContoCorrente(double saldoIniziale) { ... }
    public void versa(double somma) { ... }
    public double getSaldo() { return saldo; }
    public boolean preleva(double somma) {
        if (saldo >= somma) { ... }
        else return false;
    }
}

// al posto dei ... ci va il codice
// qui omesso per motivi di spazio!
```

Esempio: Interfaccia BankAccount

Aver aggiunto l'interfaccia non ha cambiato nulla nell'esecuzione del programma

- ▶ Ora però è **esplicito** che il conto corrente è un **tipo di dato astratto** e la sua **segnatura** è nell'interfaccia
- ▶ Indicando `implements BankAccount` la classe `ContoCorrente` è **obbligata** a implementare tutti i metodi dell'interfaccia (**il compilatore controlla!**)
- ▶ Potremmo avere altre implementazioni dell'interfaccia...

Esempio: ContoLimitato

Diamo un'altra implementazione di BankAccount che consente di fare un numero limitato di movimenti:

```
public class ContoLimitato implements BankAccount {
    private double saldo;
    private int mov; // movimenti rimasti
    public ContoCorrente(double saldoIniziale, int movimenti) {
        saldo = saldoIniziale; mov = movimenti;
    }
    public void versa(double somma) { if (mov>0) { mov--; saldo+= somma; }}
    public double getSaldo() { return saldo; }
    public int getMov() { return mov; } // in più rispetto all'interfaccia
    public boolean preleva(double somma) {
        if (saldo>=somma && mov>0) { mov--; saldo-=somma; return true; }
        else return false;
    }
}
```

Implementazioni alternative e Nominal Subtyping

Con più implementazioni alternative dell'interfaccia BankAccount, si può scegliere di volta in volta quale usare

- ▶ BankAccount si può usare come **tipo** e `implements` implica la seguente relazione di **sottotipo**

ContoCorrente <: BankAccount

ContoLimitato <: BankAccount

- ▶ Questo è un caso di **Nominal Subtyping**:
 - ▶ La relazione di sottotipo nasce dal fatto che le due classi **nominano esplicitamente** l'interfaccia BankAccount
 - ▶ Il compilatore controlla anche la **proprietà strutturale** (tutti i membri dell'interfaccia implementati nella classe) ma quello che conta è che ci sia scritto `implements BankAccount` nel codice!

Classe Banca ancora aggiornata

Quindi ora la classe Banca può scegliere...

```
public class Banca {  
    public static void main(String[] args) {  
  
        // usa due BankAccount implementati diversamente  
        BankAccount conto1 = new ContoCorrente(1000);    // un CC  
        BankAccount conto2 = new ContoLimitato(200,10);  // e un CL  
  
        if (conto1.preleva(700)) {  
            conto2.versa(700);  
        }  
  
        System.out.println("Saldo primo conto: " + conto1.getSaldo() );  
        System.out.println("Saldo secondo conto: " + conto2.getSaldo() );  
    }  
}
```

Classe Banca ancora aggiornata

Quindi ora la classe Banca può scegliere...

```
public class Banca {  
    public static void main(String[] args) {  
  
        // usa due BankAccount implementati diversamente  
        BankAccount conto1 = new ContoCorrente(1000);    // un CC  
        BankAccount conto2 = new ContoLimitato(200,10);  // e un CL  
  
        In questo caso:  
        - BankAccount è il TIPO APPARENTE (o TIPO STATICO) di conto1 e conto2  
        - ContoCorrente è il TIPO EFFETTIVO (o TIPO DINAMICO) di conto1  
        - ContoLimitato è il TIPO EFFETTIVO (o TIPO DINAMICO) di conto2  
  
        System.out.println("Saldo secondo conto: " + conto2.getSaldo() );  
    }  
}
```


Classe Banca ancora aggiornata

Quindi ora la classe Banca può scegliere

Il **TIPO APPARENTE** è quello usato dal compilatore per fare i suoi **controlli** (es. che un certo metodo si possa chiamare)

Il **TIPO EFFETTIVO** è il tipo che l'oggetto avrà a runtime e determina **quale versione del metodo** (quella di ContoCorrente o di ContoLimitato) dovrà essere eseguita (**DYNAMIC DISPATCH**)

```
public class Banca {  
    public void
```

```
        BankAccount conto1 = new ContoCorrente(1000);    // un CC  
        BankAccount conto2 = new ContoLimitato(200,10);  // e un CL
```

```
        if (conto1.preleva(700)) {  
            conto2.versa(700);  
        }
```

```
        System.out.println("Saldo primo conto: " + conto1.getSaldo());  
        System.out.println("Saldo secondo conto: " + conto2.getSaldo());
```

```
    }
```

```
}
```

Qui il metodo `preleva` si può chiamare perché è un metodo presente in `BankAccount` (tipo apparente di `conto1`)

A **tempo di esecuzione** sarà chiamata l'implementazione data in `ContoCorrente` (tipo effettivo di `conto1`)

Tipo apparente vs effettivo

Il **compilatore** fa i suoi **controlli statici** sul **tipo apparente** in quanto **il tipo effettivo potrebbe non essere noto** a tempo di compilazione

```
...  
BankAccount conto;  
if (sceltaUtente)  
    conto = new ContoCorrente(1000);  
else  
    conto = new ContoLimitato(1000,50);  
...  
conto.getMov(); // esiste getMov in conto? Non è detto...  
  
// L'unica certezza che ha il compilatore è che conto  
// contenga i metodi in BankAccount perché entrambe le  
// classi implementano l'interfaccia
```

Tipo apparente vs effettivo: coercion

In queste situazioni, si può **testare il tipo effettivo** (con un controllo a runtime) e «**forzare**» il compilatore con un **type cast** (**coercion** da supertipo a sottotipo)

```
...
BankAccount conto;
if (sceltaUtente)
    conto = new ContoCorrente(1000);
else
    conto = new ContoLimitato(1000,50);
...
if (conto instanceof ContoLimitato)    // test tipo effettivo
    ((ContoLimitato) conto).getMov();  // cast e chiamata

// il compilatore è soddisfatto e il metodo viene chiamato
// solo se l'oggetto è effettivamente del tipo giusto
```

Tipo apparente vs effettivo: coercion

Il **cast** (**coercion**) tra due classi, due interfacce o classe/interfaccia si può fare solo se tra le due esiste una relazione di sottotipo

- ▶ Cioè, **una esplicitamente implementa (o estende, tra classi) l'altra**
- ▶ **Nominal Subtyping**, non conta la struttura, conta il nome dopo `implements/extends`

La conversione da sottotipo a supertipo (**upcast**) è **implicita**: non è richiesto il cast

La conversione da supertipo a sottotipo (**downcast**) richiede un **cast esplicito**

```
ContoCorrente cc1 = new ContoCorrente(1000);  
BankAccount ba = cc1; // upcast implicito  
ContoCorrente cc2 = (ContoCorrente) ba; // downcast esplicito
```

NOTA: In OCaml era diverso... NO downcast!

Implementare più interfacce

Nulla vieta che una classe **implementi più interfacce**

- Consideriamo la seguente ulteriore interfaccia che descrive un **conto di deposito**: si può versare anche più volte e poi chiudere una volta per tutte riscattando il saldo

```
public interface DepositAccount {  
  
    public double getSaldo();  
    public boolean isOpen(); // dice se il conto è aperto  
    public void versa(double somma);  
    public double riscatta(); // riscatta e chiude  
  
}
```

Implementare più interfacce

Implementazione dell'interfaccia:

```
public class ContoDeposito implements DepositAccount {  
  
    private double saldo = 0;  
    private boolean open = true;  
  
    public double getSaldo() { return saldo; }  
    public boolean isOpen() { return open; }  
    public void versa(double somma) { if (open) saldo+=somma; }  
    public double riscatta() {  
        double res = saldo;  
        if (open) { saldo = -1; open = false }  
        return res;  
    }  
}
```

Nessun costruttore! Se ne usa uno di **default** e **senza parametri** che inizializza le variabili come da dichiarazione oppure (se la dichiarazione non prevede un assegnamento) a valori standard

Implementare più interfacce

Altra implementazione che combina le due interfacce

```
public class ContoFlessibile
    implements BankAccount, DepositAccount {    // due interfacce!

    private double saldo = 0;

    // metodi comuni alle due interfacce
    public double getSaldo() { return saldo; }
    public void versa(double somma) { saldo+=somma; }

    // metodi solo in BankAccount
    public boolean preleva(double somma) {
        if (saldo>=somma) { saldo-=somma; return true; }
        else return false;
    }

    // metodi solo in DepositAccount
    public boolean isOpen() { return true; }    // questo conto non chiude mai
    public double riscatta() {
        double res = saldo;
        saldo = 0; return res;
    }
}
```

ContoFlessibile <: BankAccount
ContoFlessibile <: DepositAccount

Membri statici e d'istanza

Abbiamo visto che il `main` è un **metodo statico**:

- ▶ Può essere richiamato anche se non sono ancora stati istanziati oggetti della classe che lo contiene
- ▶ Distinguiamo:
 - ▶ **membri (variabili e metodi) d'istanza**: codificano lo **stato** di ogni **singolo oggetto** (variabili d'istanza) e implementano operazioni che lavorano su tale stato (metodi d'istanza)
 - ▶ **membri (variabili e metodi) statici**: codificano informazioni e operazioni «**di classe**», ossia variabili **condivise** tra tutti gli oggetti che sono istanze di quella classe (variabili statiche) e metodi che **non operano sullo stato** dei singoli oggetti

Membri statici e d'istanza

Operativamente:

- ▶ Una **variabile d'istanza** è presente in memoria in tante copie quanti sono gli oggetti
- ▶ Una **variabile statica** è presente una sola volta in memoria, anche prima che sia creato il primo oggetto di quella classe
- ▶ Un **metodo d'istanza** è richiamato su un oggetto e può accedere sia alle variabili d'istanza (relative a quell'oggetto) che a quelle statiche
- ▶ Un **metodo statico** può accedere solo alle variabili statiche e può essere richiamato anche se non ci sono oggetti di quella classe

Esempio: Ancora ContoCorrente

Estendiamo la classe ContoCorrente con due funzionalità:

- ▶ La possibilità di dare ad ogni conto un **numero identificativo univoco**
 - ▶ Useremo una **variabile d'istanza** per memorizzare il **numero del conto** nel suo stato
 - ▶ E una **variabile statica** (contatore condiviso tra gli oggetti) per **generare numeri freschi** nel costruttore
- ▶ La descrizione di un **tasso di interesse** uguale per tutti i conti:
 - ▶ Useremo una **variabile statica** per dividerla tra tutti gli oggetti

```
public class ContoCorrente implements BankAccount {
    private double saldo;
    private int numero; // numero del conto corrente
    private static int numeroUltimoConto = 1000 // contatore (statico)
    private static double tasso; // tasso di interesse (statico)

    public ContoCorrente(double saldoIniziale) {
        saldo=saldoIniziale; // ad ogni creazione di un conto
        numeroUltimoConto++; // aggiorna il contatore condiviso
        numero=numeroUltimoConto;
    }

    public void versa(double somma) { ... }
    public boolean preleva(double somma) { ... }
    public double getSaldo() { return saldo; }
    public double getNumero() { return numero; }
    public static double getTasso() { return tasso; } // metodo statico
    public maturaInteressi() { saldo += saldo*tasso; } // calcolo interessi
}
```

Modello della memoria

Vediamo **che cosa succede nella memoria** della JVM eseguendo il programma della Banca

In un primo modello astratto della memoria possiamo identificare tre aree:

- ▶ L'**AMBIENTE DELLE CLASSI** (o **WORKSPACE**), che contiene il codice dei metodi e le **variabili statiche** (di classe)
- ▶ Lo **STACK**, che contiene i **record di attivazione** dei metodi con le **variabili locali**
- ▶ Lo **HEAP**, che contiene gli **oggetti** (raggiungibili tramite **riferimenti**) con le loro **variabili d'istanza**

AMBIENTE DELLE
CLASSI

STACK

HEAP

CLASS UsADueConti

main()

CLASS ContoCorrente

ContoCorrente()

VERSA()

PRELEVAC)

OTTIENI SALDO()

OTTIENI NUMERO()

MATURA INTERESSI()

NUMEROUltimoConto =

TASSO =

HEAP

STACK

AMBIENTE DELLE
CLASSI

CLASS UsADUECONTI

main()

CLASS CONTOCORRENTE

CONTOCORRENTE()

VERSA()

PRELEVA()

OTTIENI SALDO()

OTTIENI NUMERO()

MATURA INTERESSI()

NUMEROUltimoConto =

TASSO =

```
public class Banca {
    public static void main(String[] args) {
        ContoCorrente conto1 =
            new ContoCorrente(1000);
        ContoCorrente conto2 =
            new ContoCorrente(200);
        if (conto1.preleva(700)) {
            conto2.versa(700);
        }
        System.out.println("Saldo1 " + ...)
        System.out.println("Saldo2 " + ...)
    }
}
```


HEAP

STACK

AMBIENTE DELLE CLASSI

UsaDueConti.main()

← RECORD DI ATTIVAZIONE METODO main()

CLASS UsaDueConti
main()

CLASS ContoCorrente
ContoCorrente()
Versa()
Preleva()
OttieniSaldo()
OttieniNumero()
MaturaInteressi()
NumeroUltimoConto = 1000
Tasso = 0,02

← METODI

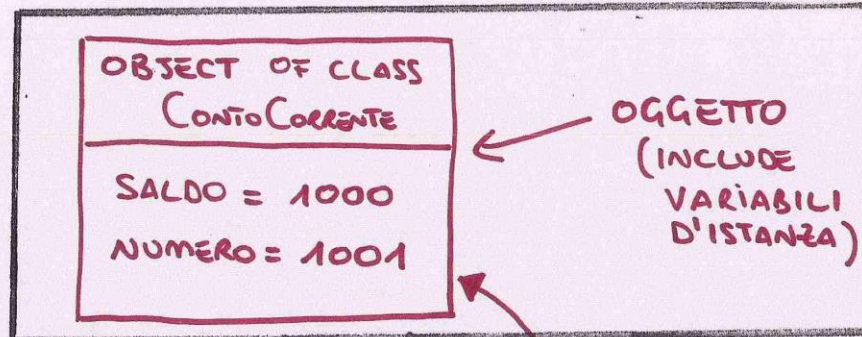
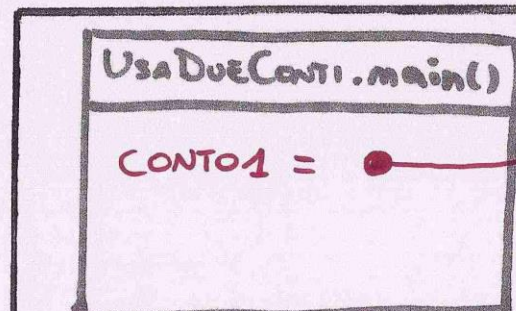
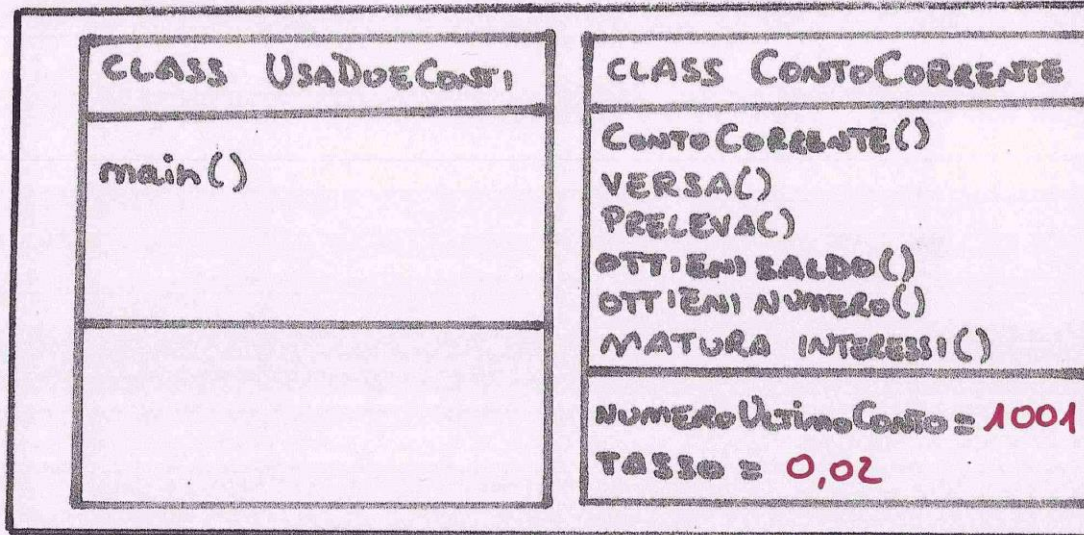
← VARIABILI STATICHE (INIZIALIZZATE)

```
public class Banca {  
    public static void main(String[] args) {  
        ContoCorrente conto1 =  
            new ContoCorrente(1000);  
        ContoCorrente conto2 =  
            new ContoCorrente(200);  
        if (conto1.preleva(700)) {  
            conto2.versa(700);  
        }  
        System.out.println("Saldo1 " + ...)  
        System.out.println("Saldo2 " + ...)  
    }  
}
```


AMBIENTE DELLE
CLASSI

STACK

HEAP



OGGETTO
(INCLUDE
VARIABILI
D'ISTANZA)

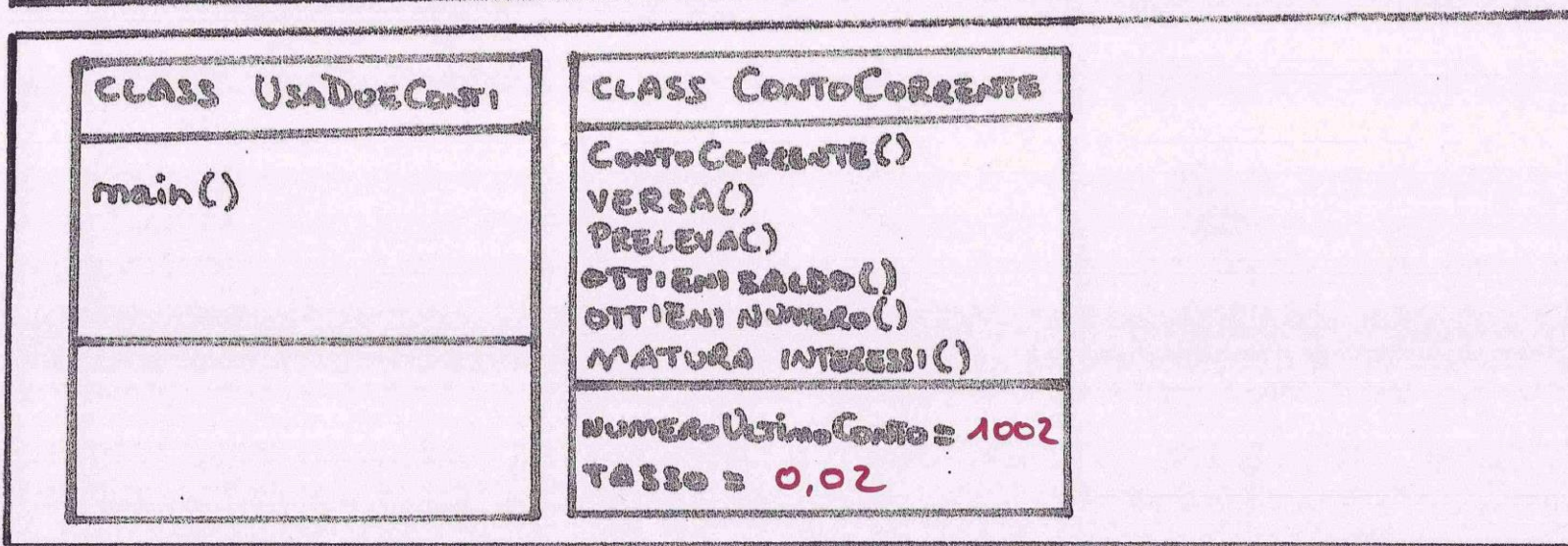
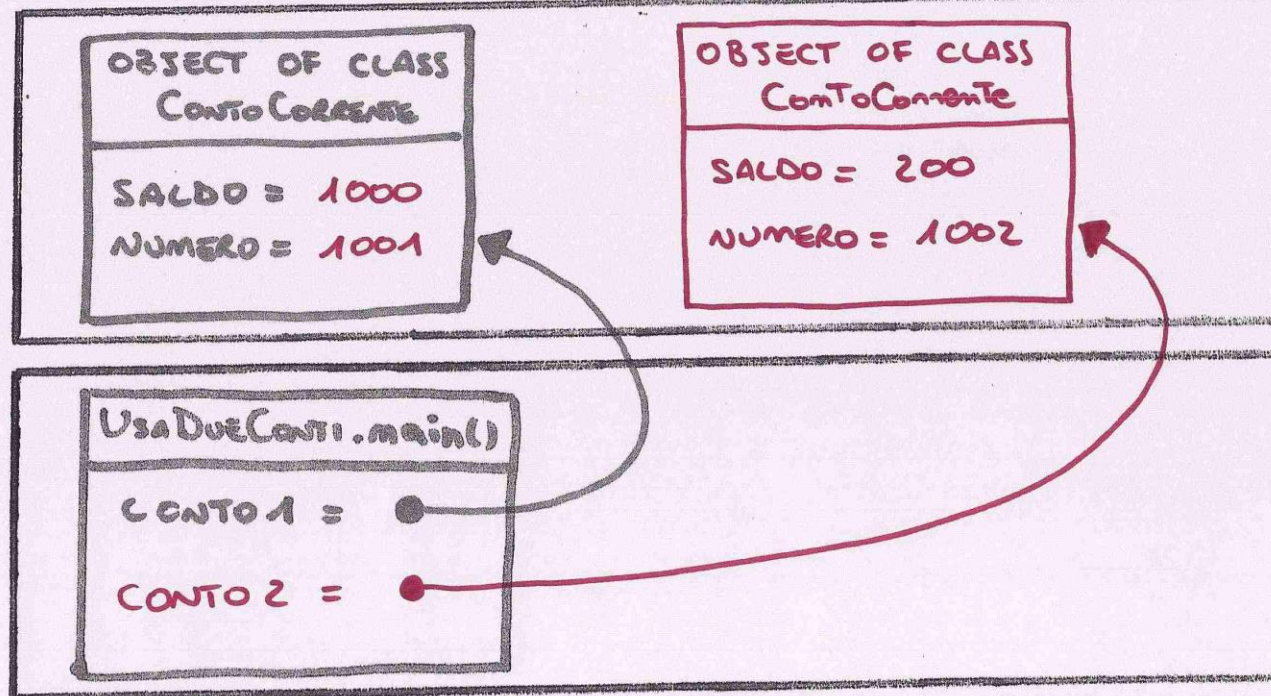
RIFERIMENTO

```
public class Banca {  
    public static void main(String[] args) {  
        ContoCorrente conto1 =  
            new ContoCorrente(1000);  
        ContoCorrente conto2 =  
            new ContoCorrente(200);  
        if (conto1.preleva(700)) {  
            conto2.versa(700);  
        }  
        System.out.println("Saldo1 " + ...)  
        System.out.println("Saldo2 " + ...)  
    }  
}
```


HEAP

STACK

AMBIENTE DELLE
CLASSI

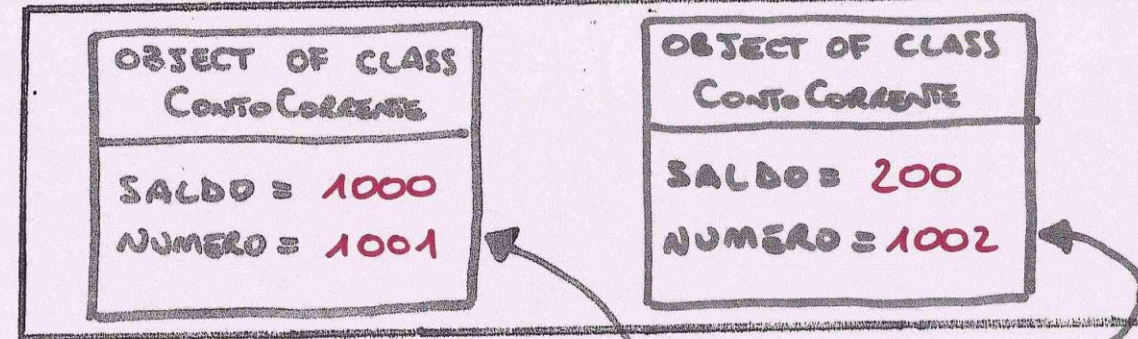
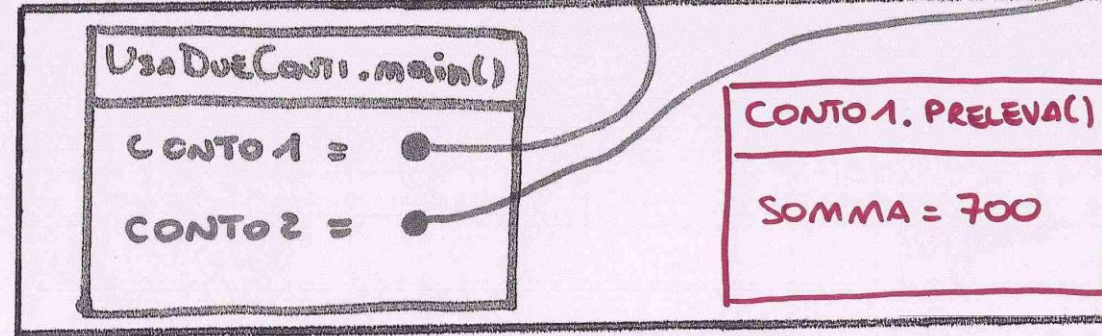
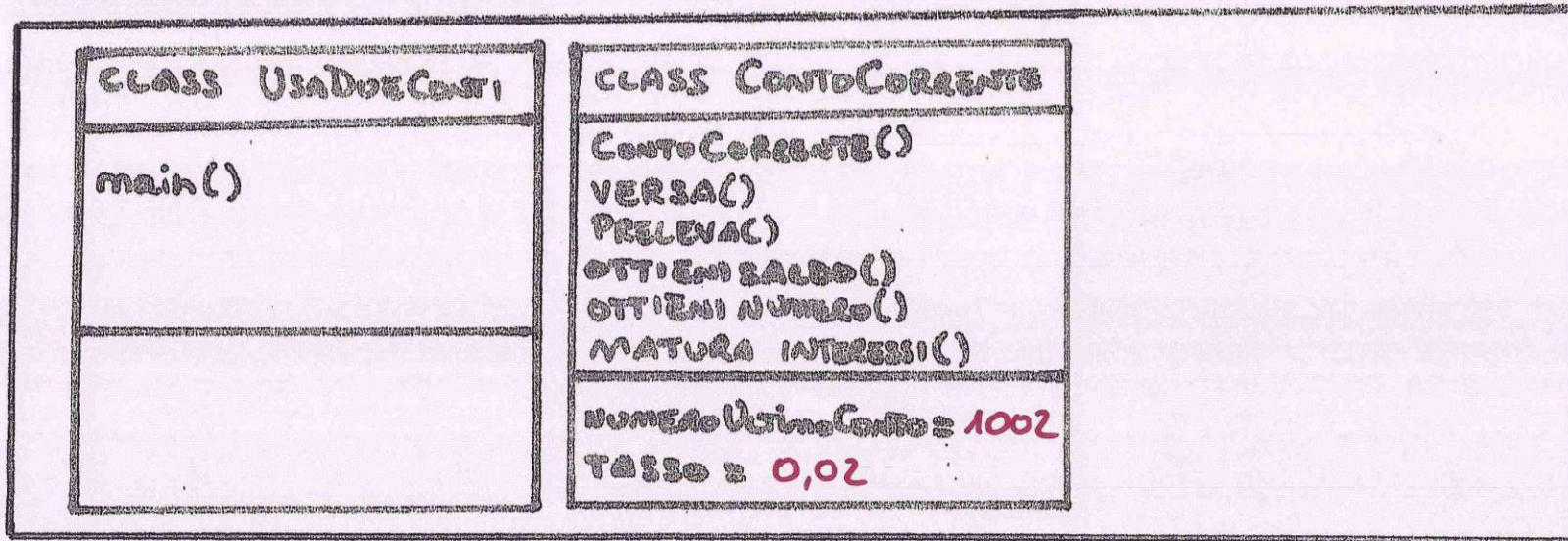


```
public class Banca {
    public static void main(String[] args) {
        ContoCorrente conto1 =
            new ContoCorrente(1000);
        ContoCorrente conto2 =
            new ContoCorrente(200);
        if (conto1.preleva(700)) {
            conto2.versa(700);
        }
        System.out.println("Saldo1 " + ...)
        System.out.println("Saldo2 " + ...)
    }
}
```


AMBIENTE DELLE
CLASSI

STACK

HEAP

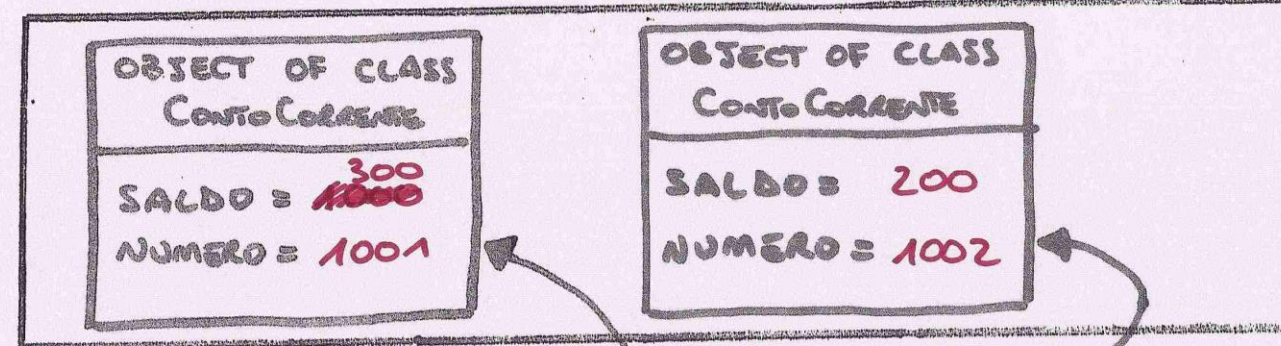
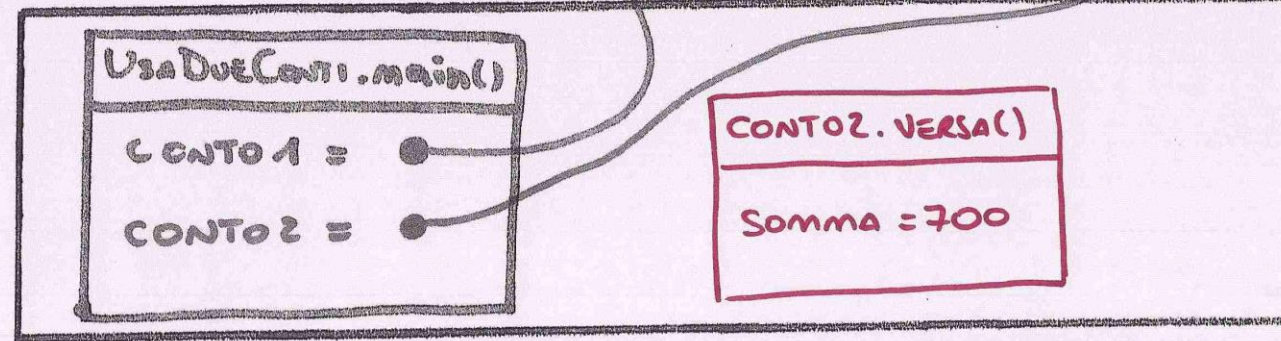
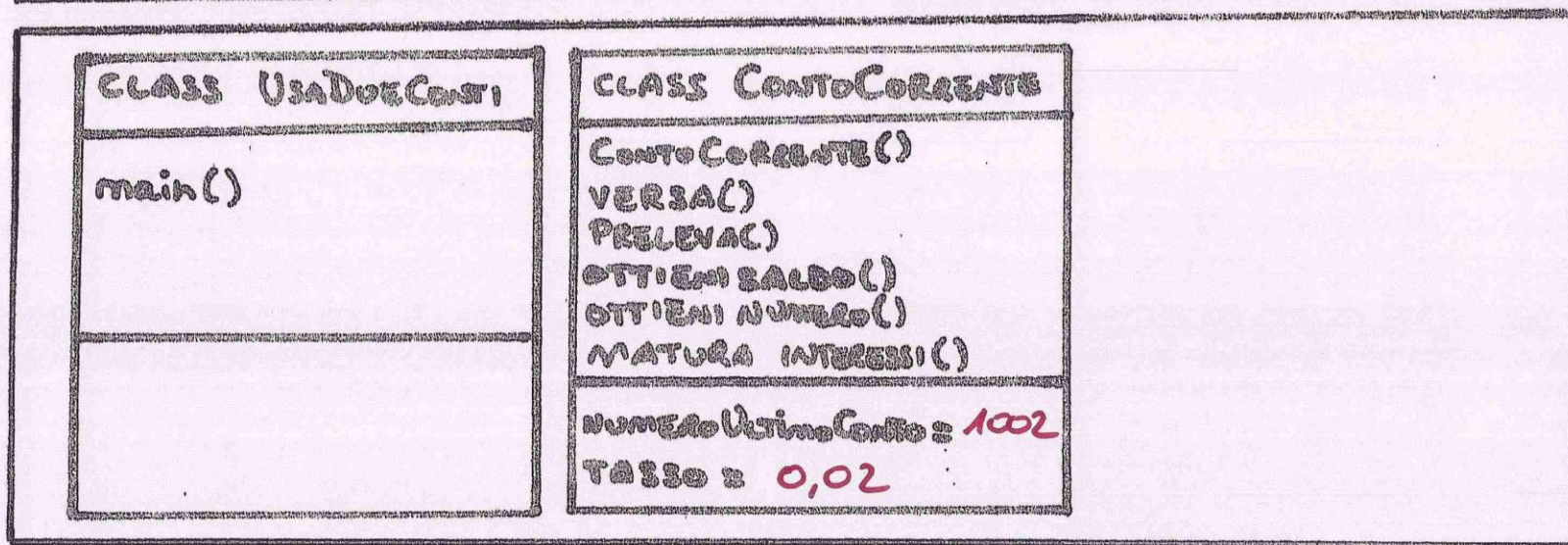


```
public class Banca {
    public static void main(String[] args) {
        ContoCorrente conto1 =
            new ContoCorrente(1000);
        ContoCorrente conto2 =
            new ContoCorrente(200);
        if (conto1.preleva(700)) {
            conto2.versa(700);
        }
        System.out.println("Saldo1 " + ...)
        System.out.println("Saldo2 " + ...)
    }
}
```


AMBIENTE DELLE CLASSI

STACK

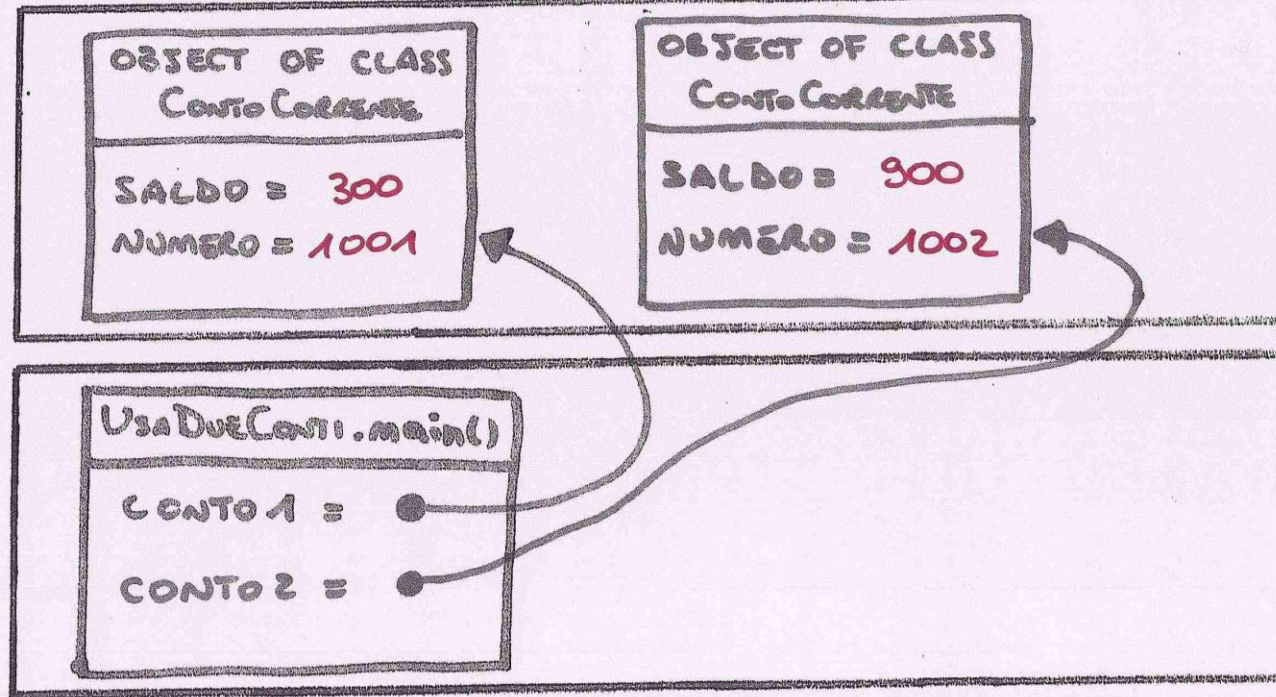
HEAD



```
public class Banca {
    public static void main(String[] args) {
        ContoCorrente conto1 =
            new ContoCorrente(1000);
        ContoCorrente conto2 =
            new ContoCorrente(200);
        if (conto1.preleva(700)) {
            conto2.versa(700);
        }
        System.out.println("Saldo1 " + ...)
        System.out.println("Saldo2 " + ...)
    }
}
```


HEAD

STACK

AMBIENTE DELLE
CLASSI

```
public class Banca {
    public static void main(String[] args) {
        ContoCorrente conto1 =
            new ContoCorrente(1000);
        ContoCorrente conto2 =
            new ContoCorrente(200);
        if (conto1.preleva(700)) {
            conto2.versa(700);
        }
        System.out.println("Saldo1 " + ...)
        System.out.println("Saldo2 " + ...)
    }
}
```

Altro esempio, lista di interi

```
class Node {  
    private int elt;  
    private Node next;
```

```
    public Node (int e0, Node n0) {  
        elt = e0;  
        next = n0;  
    }  
    :  
}
```

```
public static int m( ) {  
    Node n1 = new Node(1,null);  
    Node n2 = new Node(2, n1);  
    Node n3 = n2;  
    n3.next.next = n2;  
    Node n4 = new Node(4, n1.next);  
    n2.next.elt = 17;  
    return n1.elt;  
}
```

Altro esempio, lista di interi

- ▶ Supponiamo di voler valutare l'invocazione
 - `Node.m();`
- ▶ La prima cosa da osservare è che l'invocazione del metodo restituisce un valore intero (che non è un oggetto)
- ▶ Lo stack deve contenere lo spazio per memorizzare il valore restituito dall'invocazione del metodo
 - intuitivamente una variabile di tipo `int`
 - per semplicità lo omettiamo

WORKSPACE

```
n3.next.next = n2;  
Node n4 = new Node (4, n1.next);  
n2.next.elc = 17;  
return n1.elc
```

STACK

n1	
n2	
n3	

HEAP

NODE

elt	1
next	null

NODE

elt	2
next	

```
Node n1 = new Node (1, null);  
Node n2 = new Node (2, n1);  
Node n3 = n2;
```

WORKSPACE

```
n3.next.next = n2;  
Node n4 = new Node (4, n1.next);  
n2.next.elc = 17;  
return n1.elc
```

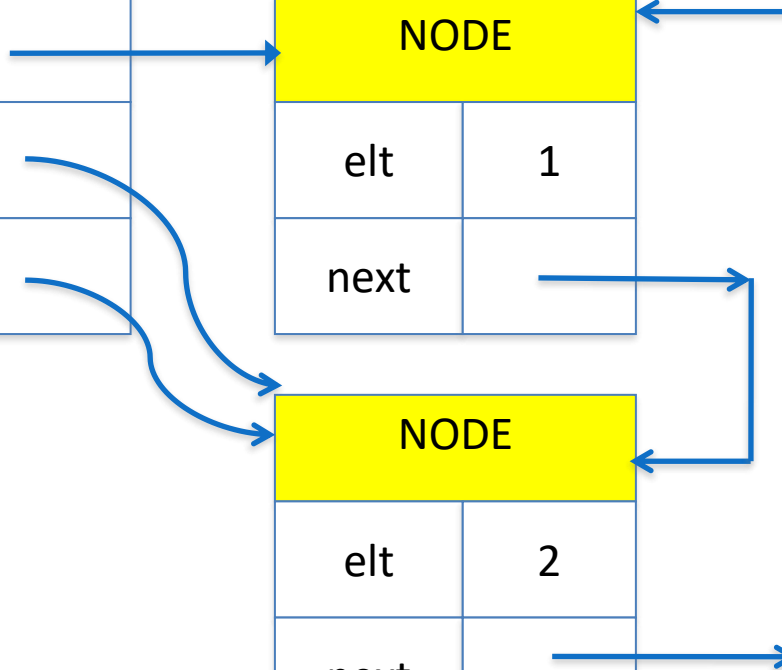
STACK

n1	
n2	
n3	

HEAP

NODE	
elt	1
next	

NODE	
elt	2
next	



WORKSPACE

```
Node n4 = new Node (4, n1.next);  
n2.next.elc = 17;  
return n1.elc
```

STACK

n1	
n2	
n3	
n4	

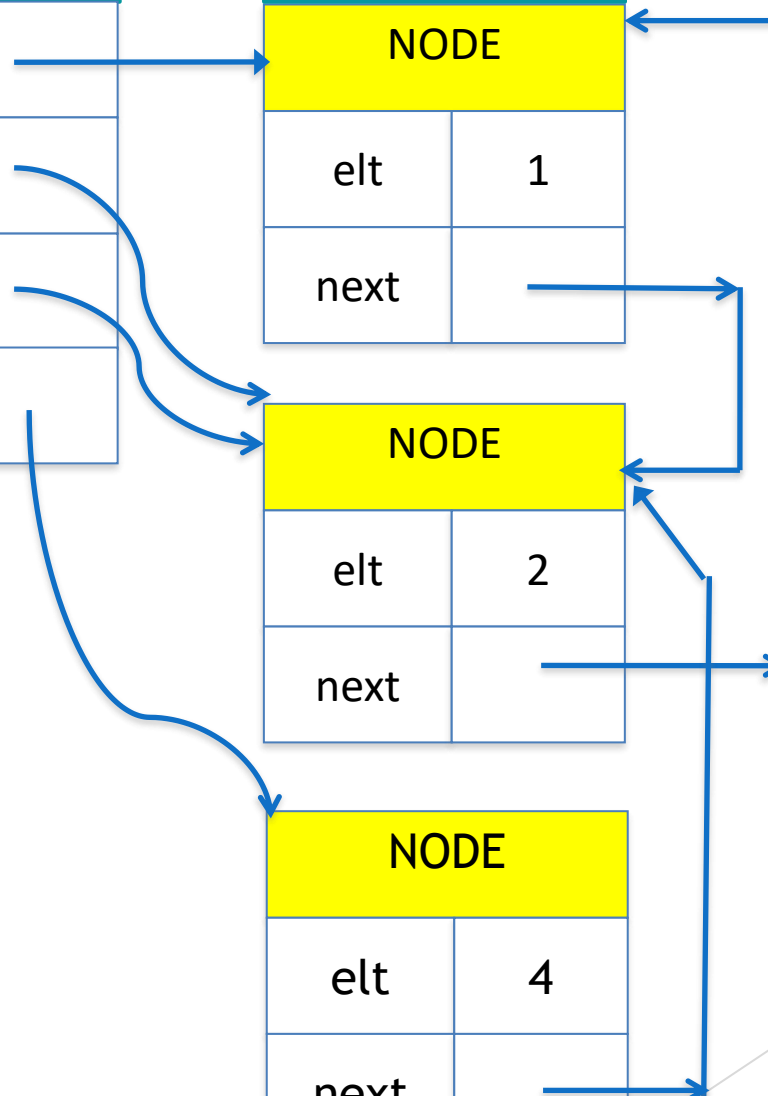
HEAP

NODE	
elt	1
next	

NODE	
elt	2
next	

NODE	
elt	4
next	

Domanda: quale è il valore restituito?



WORKSPACE

```
Node n4 = new Node (4, n1.next);  
n2.next.elt = 17;  
return n1.elt
```

STACK

n1	
n2	
n3	
n4	

HEAP

NODE

elt	1
next	

NODE

elt	2
next	

NODE

elt	4
next	

Domanda: quale è il valore restituito? **17**
Come mai?

Riferimenti

Gli oggetti (creati con new) diventano accessibili tramite **riferimenti** memorizzati nelle variabili locali (o in altri oggetti raggiungibili... es. liste di oggetti)

Il meccanismo dei riferimenti agli oggetti funziona come in JavaScript...

- ▶ Possono verificarsi situazioni di **ALIASING**! (come in es. precedente)
- ▶ Un oggetto può rimanere orfano (non essere più raggiungibile tramite riferimenti). In questo caso interviene il **Garbage Collector** a liberare la memoria (vedremo come)