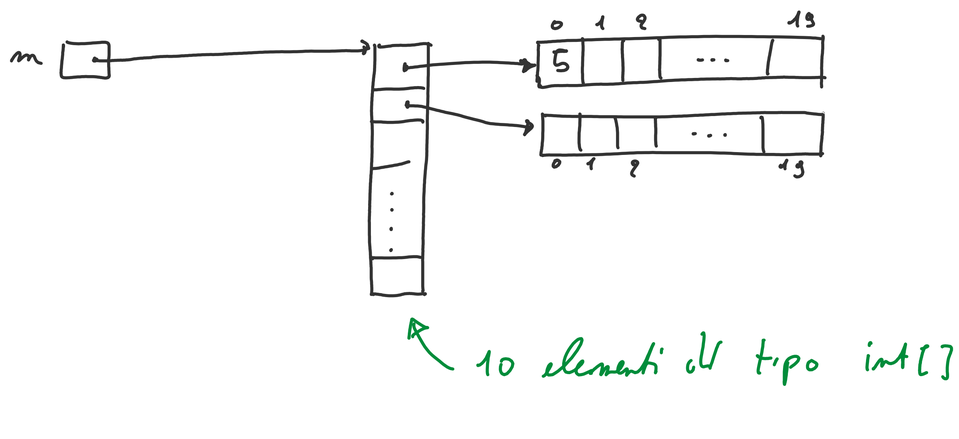
Lezione 5

giovedì 18 ottobre 2018 15:42

Array multidimensionali

```
int[][] m;
m = new int[10][20];
```



10 elementi di tipo int[]

Per accedere agli elementi:

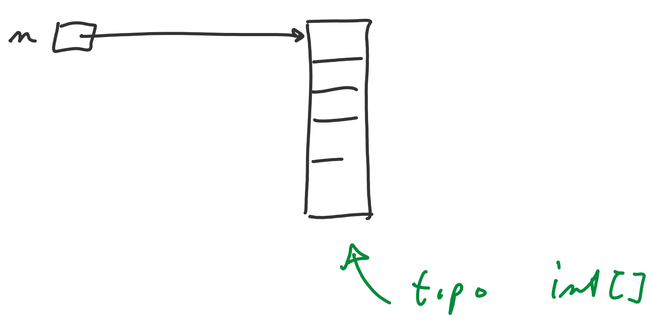
```
m[0][0] = 5;
```

Altro esempio:

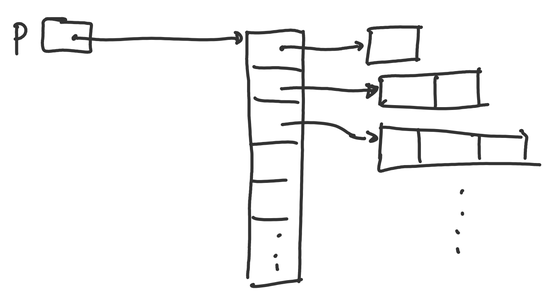
```
for (int i=0; i<m.length; i++)
    for (int j=0; j<m[i].length; j++)
        m[i][j] = ....
```

- La dimensione dei sub-array può essere non specificata

```
int[][] m = new int[5][];
```




```
int[][] p = new int[10][];
for (int i=0; i<p.length; i++)
    p[i] = new int[i+1];
```



Metodi di utilità

classe System:

```
static void arrayCopy (Object from, int fromIndex,
```

Object to, int toIndex,
int count)

Copia dall'array from all'array to
partendo dagli indici fromIndex e toIndex rispettivamente
count è il numero di elementi da copiare

classe Arrays (package java.util)

static void sort(PrimitiveType[] a)

static int binarySearch(PrimitiveType[] a,
PrimitiveType v)

Metodo main

- Punto di ingresso per l'esecuzione del programma
- La signature è

  public static void main(String[] args)

- Non c'è la versione priva di argomenti
- args contiene i valori (stringhe) specificati da riga di comando

```
public class Prova {
    public static void main(String[] args) {
        for (int i=0; i< args.length; i++)
            System.out.println(args[i]);
    }
}
```

$ java Prova abc def ghi

args[0] ⟶ "abc"

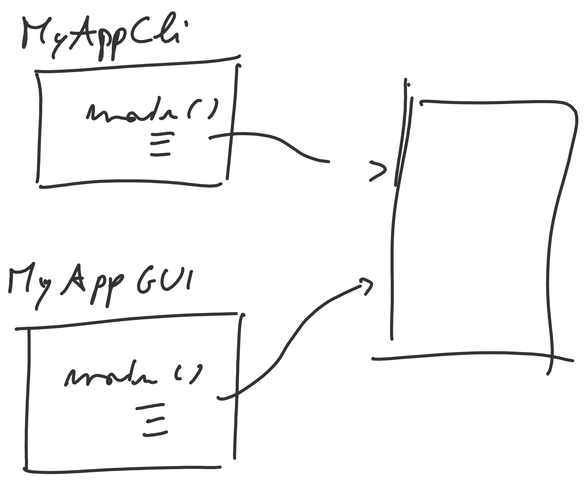
args[1] ⟶ "def"

args[2] ⟶ "ghi"

- Più classi della stessa applicazione possono essere dotate di metodo main
  - non è un problema perché da riga di comando dobbiamo sempre specificare il nome della classe da mandare in esecuzione.

A cosa serve?

- esempio: applicazione che può interagire con l'utente sia mediante GUI che da riga di comando (CLI)



- Main "da prova"

```
class X {
    ≡
    void metodocomplicato(){
        ≡
    }
    public static void main(String[] args){
        X x = new X();
        x.metodocomplicato();
        ≡
    }
}
class Y {
    ≡
    public static void main(String[] args){
        Y y = new Y();
        y. ...
    }
}
class Z {
    ≡
    public static void main(String[] args) {
        // main vero e proprio
        ≡
    }
}
```

Ereditarietà

Permette di definire nuove classi a partire da classi esistenti (specializz., per differenza).

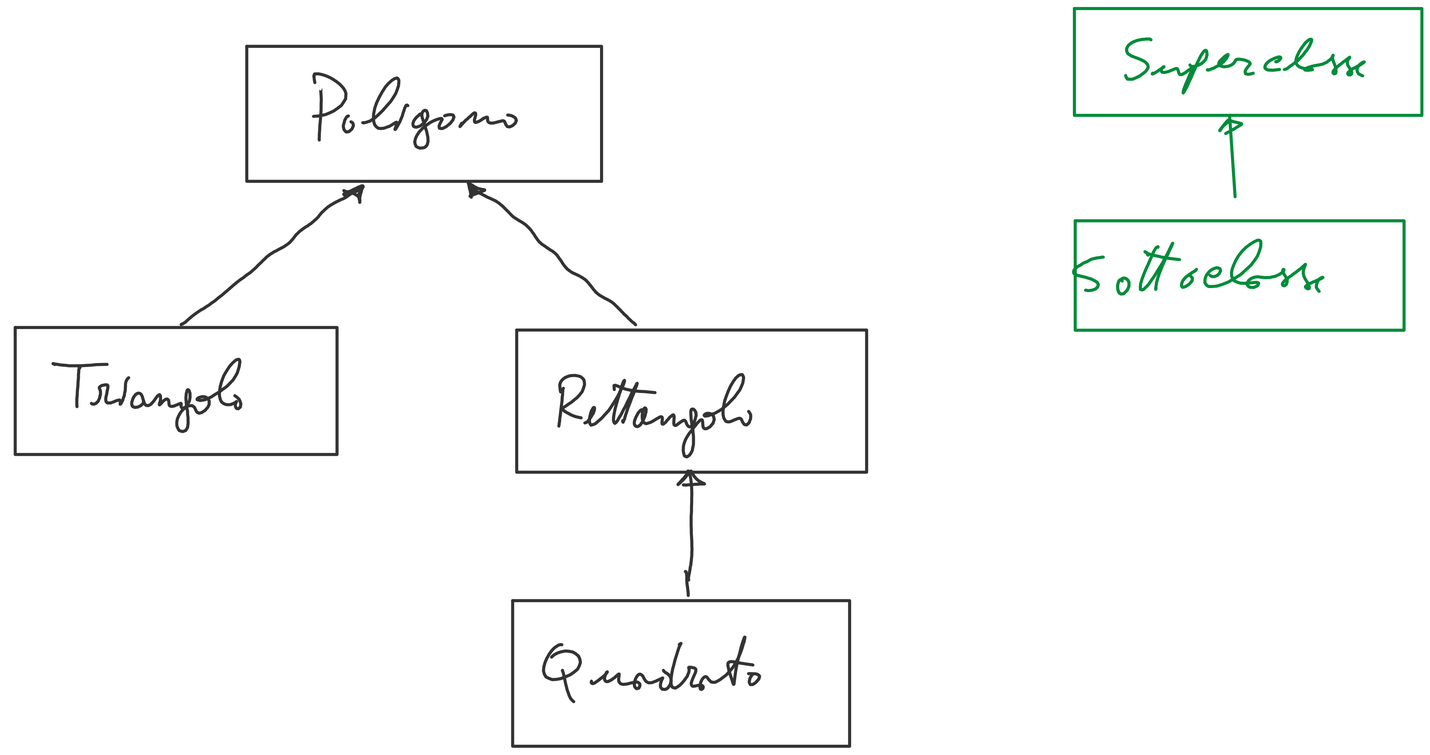
```
class A {
    ≡
}

class B extends A {
```

A: supertipo, classe base, classe padre, superclasse

B: sottotipo, classe derivata, classe figlio, sottoclasse

}

- In Java l'ereditarietà è singola
  - una classe può avere una sola superclasse
- Una classe può avere più sottoclassi
- È possibile generare gerarchie con più livelli



La relazione sottoclasse – superclasse è del tipo "è un" ("is a")

Un Triangolo è un Poligono

Un Rettangolo è un Poligono

Un Quadrato è un Rettangolo e quindi anche un Poligono.

Non vale l'inverso:
un Poligono non è necessariamente un Triangolo

Con extends abbiamo contemporaneamente
- ereditarietà del contratto* della superclasse
- ereditarietà dell'implementazione

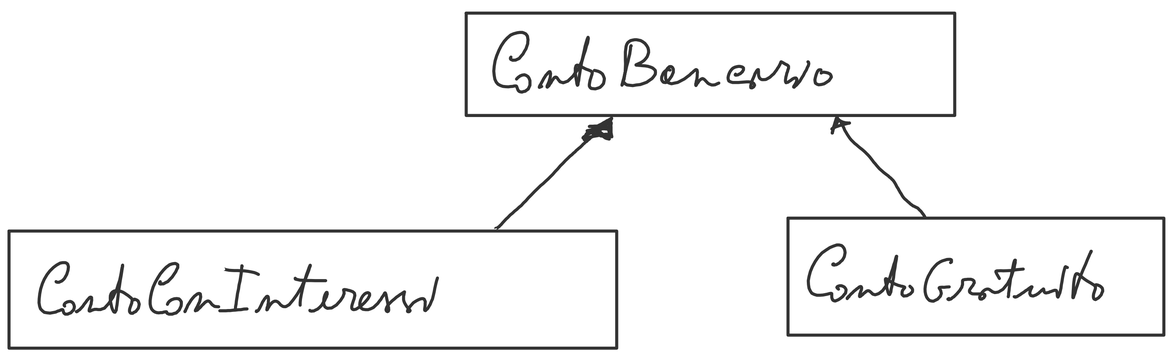
Contratto = insieme di operazioni che è possibile eseguire su un determinato tipo

- Perché ereditarietà ? Riuso del codice

- In una sottoclasse è possibile
  - accedere ai campi e ai metodi della superclasse
  - aggiungere nuovi campi e nuovi metodi
  - ridefinire il comportamento dei metodi della superclasse.

- Non è possibile accedere ai campi e ai metodi privati della superclasse.

- Realizziamo la seguente gerarchia:



Le operazioni possibili su un conto bancario sono:

- deposito
- prelievo
- trasferimento
- getProprietà (metodi getter)
- toString

```
public class ContoBancario {
    private double bilancio;
    private final String intestatario;
    private final int numero;
    private static int prossimo = 1;

    public ContoBancario(String i) {
        intestatario = i;
        numero = prossimo++;
    }

    public String getIntestatario() {
        return intestatario;
    }

    public double getBilancio() {
        return bilancio;
    }
```

```
public int getNumero(){
    return numero;
}

public void deposita(double d){
    bilancio += d;
}

public void preleva(double d){
    bilancio -= d;
}

public void trasferisci(ContoBancario c, double d){
    preleva(d);
    c.deposita(d);
}

public String toString(){
    return "intestatario = " + intestatario +
           "  bilancio = " + bilancio +
           "  numero = " + numero;
}
}
```

Conto con interessi è un conto bancario caratterizzato da un tasso, frutta interessi = tasso * bilancio

- aggiungiInteressi
- getTasso

[eredita deposita, trasferisci, preleva
  modifica toString]

```
public class ContoConInteressi extends ContoBancario {
    private double tasso;

    public ContoConInteressi(String i, double t){
        super(i);
        tasso = t;
    }
```

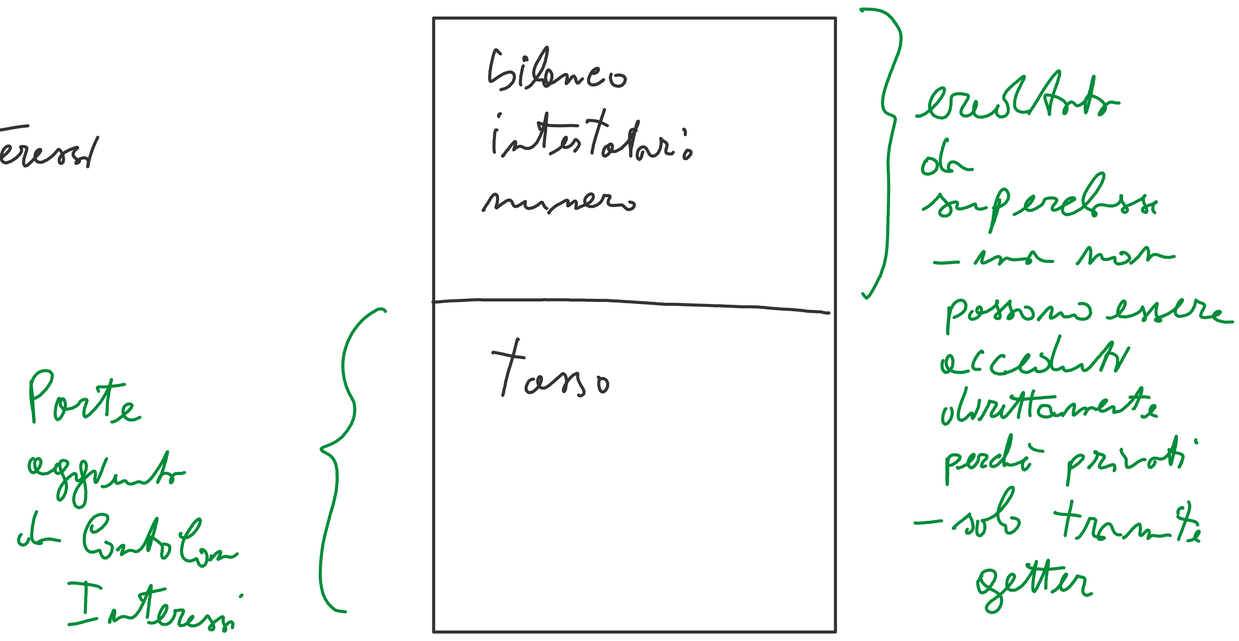
richiama il costruttore di ContoBancario.
Deve essere la prima istruzione del costruttore della classe derivata.

```
public void aggiungiInteressi(){
    double interessi = getBilancio() * tasso;
    deposita(interessi);
}

public String toString(){
```

```
        String r = super.toString();
        r += "tasso" + tasso;
        return r;
    }
}
```

vogl o chiamare il metodo toString della superclasse
- se non lo mettessimo il metodo toString richiamerebbe se stesso (ricorsivo).

ContoConInteressi



Una sottoclasse deve richiamare il costruttore della superclasse attraverso super.

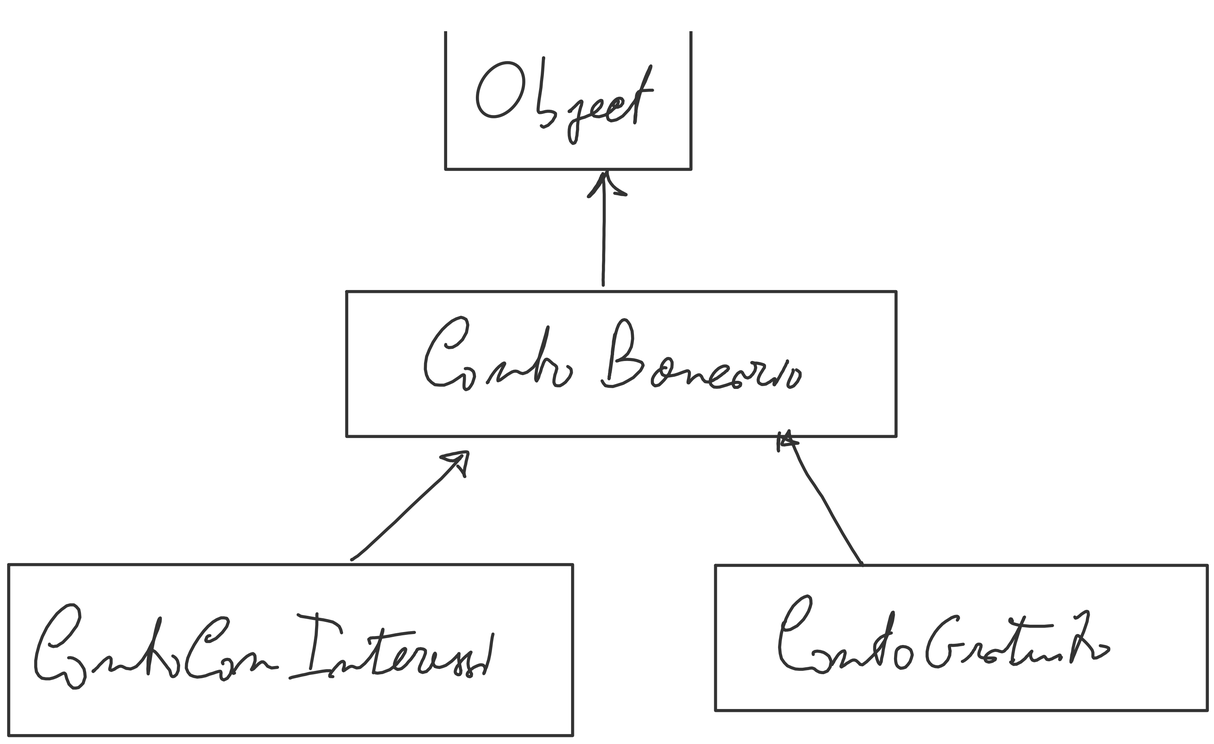
- il costruttore della superclasse inizializza la sezione superclasse

- La chiamata a super() deve essere la prima istruzione del costruttore della classe derivata
  - eltre istruzioni a seguire, se necessario

- Se la superclasse ha un costruttore privo di argomenti (per es. il costruttore di default) allora la chiamata a super() può essere omessa (la aggiunge il compilatore).

- Se la superclasse ha solo costruttori con argomenti la sottoclasse deve indicare quale usare richiamare super (arg. del costr. da richiamare)

Una classe che non ne estende esplicitamente un'altra, è figlia della classe Object



Tutte le classi Java sono Object

La classe Object

Tutte le classi Java ereditano i metodi della classe Object

I metodi della classe Object possono essere divisi in
- metodi di carattere generale
- metodi per la sincronizzazione tra thread (più avanti)

Metodi di carattere generale

public boolean equals(Object o)

- confronta l'oggetto implicito con l'oggetto o restituisce true se sono uguali, false altrimenti

- il concetto di "uguale" varia da classe a classe
  - le classi ridefiniscono il metodo equals secondo il loro criterio di uguaglianza.
  - abbiamo già visto come la classe String ridefinisca equals in modo da confrontare il contenuto (caratteri) delle stringhe.

- L'implementazione di equals della classe Object esegue un confronto con == (confronto i riferimenti): true se l'oggetto implicito e l'oggetto o sono lo stesso oggetto, false altrimenti

Quindi se X non ridefinisce equals:

```
X x1 = new X();
X x2 = new X();
boolean b = x1.equals(x2); // equivale a scrivere
                           x1 == x2
```

public int hashCode()

restituisce un hashcode (valore univoco) per ogni oggetto Java.

public String toString()

restituisce una rappresentazione in formato stringa dell'oggetto a cui è applicato.
L'implementazione predefinita restituisce

nomedellaclasse@hashcode_in_esadecimale

toString() viene richiamato automaticamente quando concateniamo un oggetto a una stringa

```
ContoBancario c = new ContoBancario("Mario");
String s = "Conto: " + c;
```

equivale a scrivere

```
"Conto: " + c.toString()
```

Anche il metodo finalize che abbiamo già visto viene messo a disposizione dalla classe Object.

Riferimenti

- Un riferimento di tipo superclasse può puntare a oggetti di tipo sottoclasse

```
ContoBancario cb;

ContoConInteressi cci = new ContoConInteressi("Mario", 0.1);
cb = cci; // OK: un ContoConInteressi è
          un ContoBancario
```

```
ContoBancario cb1;
cb1 = new ContoConInteressi("Gino", 0.05);

ContoBancario cb2 = new ContoConInteressi("Furio", 0.08);

Object o1 = new ContoConInteressi("Sera", 0.09);
```

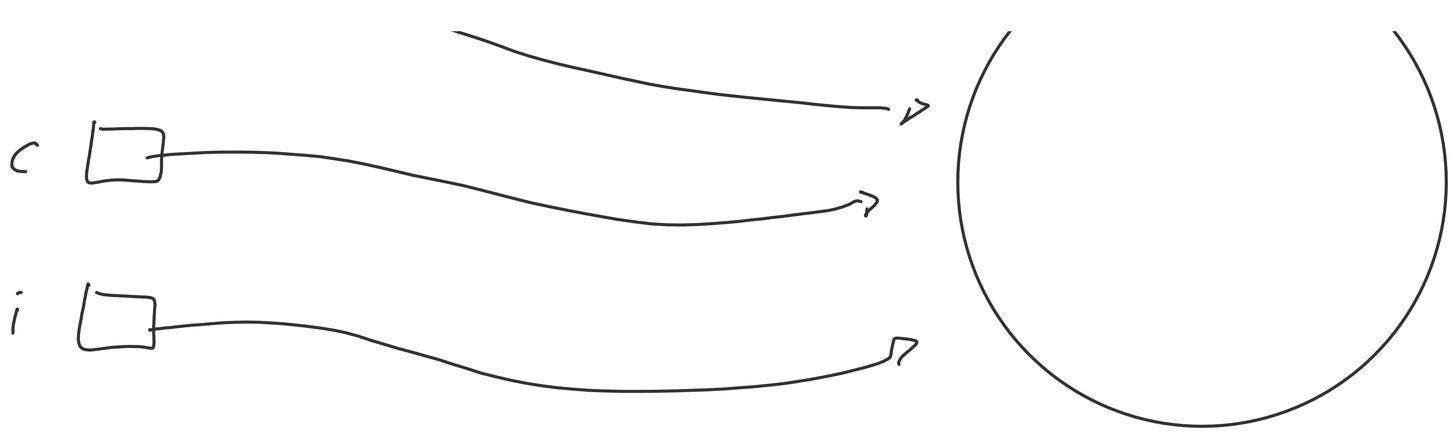
- Un supertipo può essere convertito in un sottotipo mediante un cast (tipo)
  - la conversione è possibile se l'oggetto appartiene effettivamente al tipo indicato nel cast
  - se la conversione non è possibile viene generata una ClassCastException

```
ContoConInteressi cci = new ContoConInteressi("Marco", 0.1);

ContoBancario cb;
cb = cci;
ContoConInteressi altro;
altro = (ContoConInteressi) cb; // OK: l'oggetto puntato da
                                //  cb è effettivamente un
                                //  ContoConInteressi

ContoBancario cb1 = new ContoBancario("Pino");
altro = (ContoConInteressi) cb1; // Errore: ClassCastException
                                 //  l'oggetto puntato da cb1
                                 //  è un ContoBancario e
                                 //  non è un ContoConInteressi
                                 //  (o un suo sottotipo)
```

Altro esempio:

```
Object o = new ContoConInteressi("Gino", 0.1); // OK

ContoBancario c = (ContoBancario) o;           // OK

ContoConInteressi i = (ContoConInteressi) o;   // OK
```

Quello che conta è il tipo reale dell'oggetto


c
i