Ancora sulle Classi in C++ e Java

- **this**
- **private:** vs **public:**
- inclusione librerie
- funzioni di utilità
- costruttori
- distruttori

this

- **this**
- **private:** vs **public:**
- inclusione librerie
- funzioni di utilità
- costruttori
- distruttori

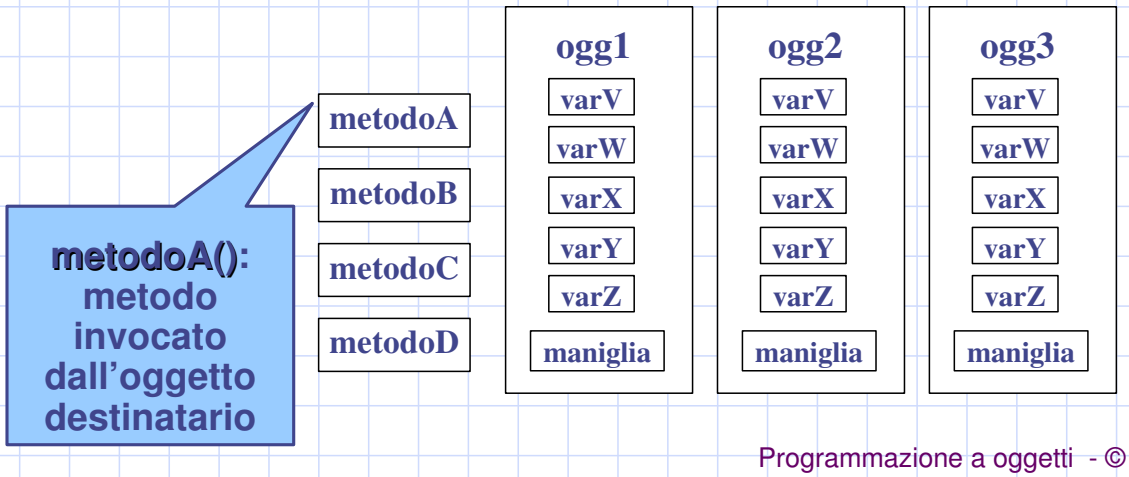
Classe: *riepilogo gestione memoria*

Ipotesi: ogg1.metodoA(ogg3)
ogg2.metodoA(ogg3)

ogg1: oggetto di invocazione del metodo

ogg3: parametro fornito al metodo

Nel primo caso metodoA() può manipolare le strutture dati di ogg1 e ogg3; nel secondo caso, quelle di ogg2 e ogg3.



metodoA(): metodo invocato dall'oggetto destinatario

Il puntatore **this** in C++

```
class Pair {
private:
      int first, second;
public:
        Pair() { first=second=0; }
        Pair( int a ) {first=second=a; }
        Pair( int a, int b ) { first=a; second=b; }
   int getFirst() { return first; }
   int getSecond() { return secon
   Pair sum( const Pair & c ) {
    Pair ris;
    ris.first = this->first + c.first;
    ris.second =  cond + c.second;
    return ris; }
};
```

componente first dell'oggetto di invocazione

componente first del parametro c

componente first dell'oggetto ris

Il riferimento **this** in Java

```
public class Pair {
    private int first = 0;
    private int second = 0;

    public Pair() {}
    public Pair(int a) {first=a; second=a;}
    public Pair(int a, int b) {first=a; second=b;}

    public int getFirst(){ return first
    public int getSecond(){ return
    public Pair sum(Pair c) {
        Pair ris = new Pair();
        ris.first = this.first + c.first;
        ris.second = second + c.second;
        return ris;}
}
```

componente first dell'oggetto di invocazione

componente first del parametro c

componente first dell'oggetto ris

La parola chiave **this**: *definizione*

- è un puntatore ad oggetti in C++ e un riferimento in Java
- esiste solo all'interno dell'implementazione dei metodi
- fa riferimento all'oggetto di attivazione, ovvero all'oggetto che ha richiesto l'attivazione del metodo
- Si usa per:
  - accedere alle strutture dell'oggetto di invocazione
  - inviare messaggi all'oggetto di invocazione
  - restituire (da parte di un metodo) l'oggetto di invocazione

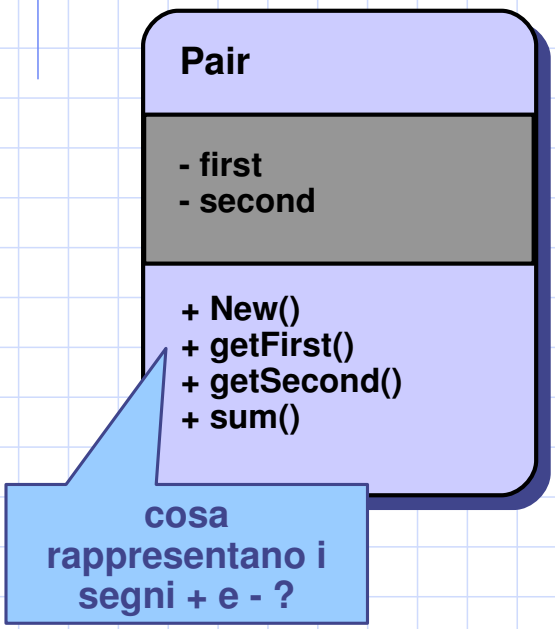
private: vs public:

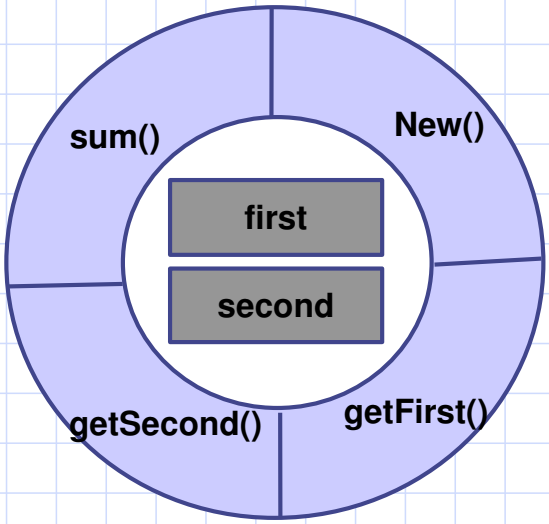
- **this**
- **private:** vs **public:**
- inclusione librerie
- funzioni di utilità
- costruttori
- distruttori

La classe "Pair": *information hiding*

Incapsulamento ed Information Hiding:

- ogni attributo di Pair non è accessibile direttamente dall'esterno





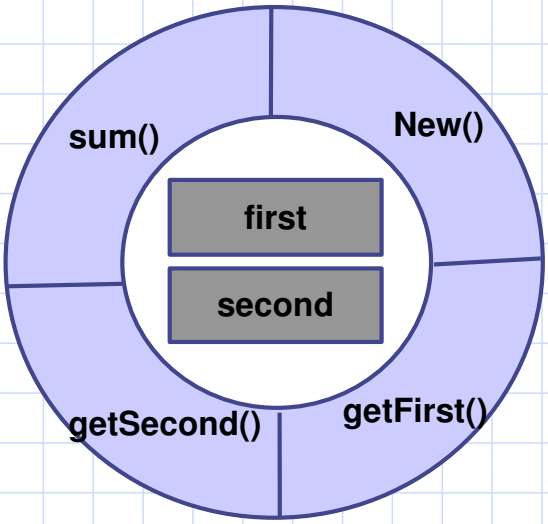
La classe “Pair”: information hiding parziale

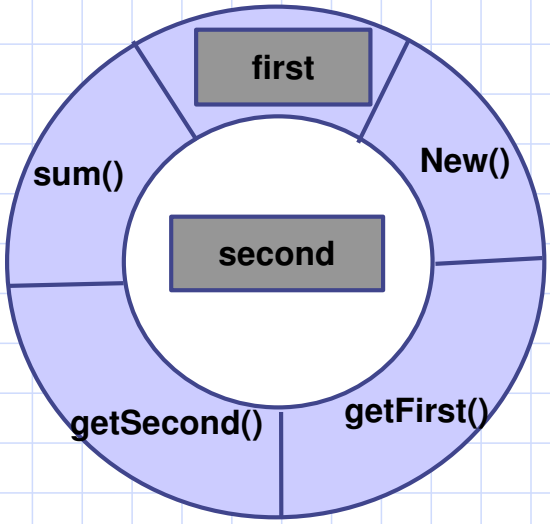
Incapsulamento ed Information Hiding:

- ogni attributo di Pair non è accessibile direttamente dall’esterno

Incapsulamento ed Information Hiding **parziale**:

- l’attributo first è accessibile direttamente dall’esterno senza passare per i metodi!





Information hiding totale e parziale: *esiste in C++*

Information hiding totale

```
class Pair {
private:
        int first, second;
public:
         Pair( int a, int b ) { … }
    int getFirst() { … }
     int getSecond() { …  }
     Pair sum( Pair c ) { … }
};
```

Information hiding parziale

```
class Pair {
private:
        int second;
public:
         Pair( int a, int b ) { … }
    int getFirst() { … }
     int getSecond() { …  }
     Pair sum( Pair c ) { … }

    int first;
};
```

L’attributo è ora accessibile direttamente dall’esterno !!

Information hiding totale e parziale: *uso*

```
// C++

main () {
 Pair x(2,4);
 cout << x.first;
 ...
}
```

```
//Java
...
public static void
main (String[] args) {
Pair x= new Pair (2,4);
System.out.print(x.first);
    ...
}
```

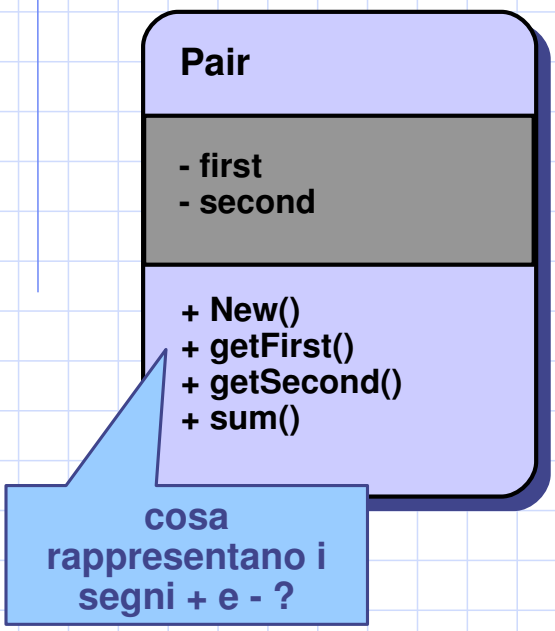
OK se first è nella sezione **public** di Pair.

Errore se first è nella sezione **private** di Pair.

Information hiding totale e parziale: *con puntatori*

```
#include "Pair.h"
main () {
   Pair * x = new Pair(2,4);
   cout << x->first;
            // OK se first è nella
            // sezione public di Pair.
            // Errore se first è nella
            // sezione private di Pair.
   ...
   delete x;
}
```

information hiding: *notazione grafica*



- I simboli + e - (nella notazione grafica ed in UML) coincidono con i concetti di elementi **private** e **public** nella definizione di classe.
- Oltre a + e -, esiste anche il simbolo #. Coincide con il concetto di elemento **protected**, ma questo concetto è usato solo nell'ambito dell'ereditarietà.

Inclusione librerie

- **this**
- **private:** vs **public:**
- inclusione librerie
- funzioni di utilità
- costruttori
- distruttori

Inclusione di librerie e file in C++

```
#include "Pair.h"

main () {
   Pair * x = new Pair(2,4);
...
}
```

Questa è una direttiva al preprocessore: indica di considerare il file Pair.h come parte integrante del file attuale

Interfaccia della classe Pair (C++)

```
// Pair.h - Interfaccia classe Pair
#ifndef PAIR_H
#define PAIR_H

class Pair {
  private:
     int first, second;
  public:
     Pair ();
     Pair ( int a );
     Pair ( int a, int b );
     int getFirst();
     int getSecond();
     Pair sum(const Pair & c);
};
#endif
```

Questa direttiva impedisce di compilare il file Pair.h più volte

- In C++ prima della compilazione viene eseguita una pre-elaborazione del codice per mezzo di un programma chiamato preprocessore
- Il preprocessore esegue istruzioni speciali chiamate direttive al preprocessore
- Le direttive al preprocessore indicano alcune manipolazioni da effettuare sul codice prima della compilazione
- Tipiche manipolazioni sono: inclusione di file esterni, sostituzione di stringhe, definizioni di simboli, etc

Implementazione metodi della classe Pair in C++

```
// Pair.cpp – Metodi della classe Pair
# include “Pair.h”

Pair::Pair () { first=second=0;
Pair::Pair ( int a ) {first=second=a; };
Pair::Pair ( int a, int b ) { first=a; second=b; };

int Pair::getFirst() { return first; };

int Pair::getSecond() { return second;  };

Pair Pair::sum(const Pair & c) {
      Pair ris;
      ris.first = first + c.first;
      ris.second = second + c.second;
      return ris; };
```

Questa è una direttiva al preprocessore

Inclusione di classi in Java

```
import java.awt.Rectangle;


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

    Rectangle box = new Rectangle(5,7,10,12);

    box.translate(15,25);
}
```

Questa NON è una direttiva al preprocessore

funzioni di utilità

- **this**
- **private:** vs **public:**
- inclusione librerie
- funzioni di utilità
- costruttori
- distruttori

TDA "*fraction*"

nome fraction
interfaccia

- *fraction (int, int) : fraction*
  precondizioni e postcondizioni per *fraction(n,m) = n'/m'*
  pre: $m \neq 0$
  post: *n/m= n'/m'*; *m'* > 0; *n' e m'* sono primi fra loro

- *getNum : (fraction) --> int*
  precondizioni e postcondizioni per *getNum(n/m) = n*
  pre: nessuna
  post: nessuna

- *isPos : (fraction) --> bool*
  precondizioni e postcondizioni per *isPos(n/m) = b*
  pre: nessuna
  post: *b=true* se e solo se *n*>0 *(ricorda che, essendo n/m una fraction, allora m>0)*

… continua

- *sum (fraction,fraction) : fraction*
  precondizioni e postcondizioni per *sum(n/m, p/q) = r/s*
  pre: nessuna
  post: *r/s = n/m + p/q*; *r* e *s* sono primi fra loro

- *sub (fraction,fraction) : fraction*
  precondizioni e postcondizioni per *sub(n/m, p/q) = r/s*
  pre: nessuna
  post: *r/s = n/m - p/q* ; *r* e *s* sono primi fra loro

- *mul (fraction,fraction) : fraction*
  precondizioni e postcondizioni per *mul(n/m, p/q) = r/s*
  pre: nessuna
  post: *r/s = n/m* × *p/q* ; *r* e *s* sono primi fra loro

- *div (fraction,fraction) : fraction*
  precondizioni e postcondizioni per *sub(n/m, p/q) = r/s*
  pre: $p \neq 0$
  post: *r/s = n/m / p/q* ; *r* e *s* sono primi fra loro

considerazioni sul progetto della classe

- Dopo aver controllato m≠0 e imposto m>0, il costruttore deve *ridurre* la frazione
- È possibile implementare facilmente le 4 operazioni sum, sub, mul e div, ma il loro risultato deve poi essere *ridotto*

Quindi:

> Almeno in 5 metodi si ripetono le stesse operazioni (riduzione di frazione). È conveniente incapsulare tali operazioni in una funzione di utilità

Ma come e dove deve essere posta questa funzione?

(vedere esempio classe fraction)

costruttori

- **this**
- **private:** vs **public:**
- inclusione librerie
- funzioni di utilità
- costruttori
- distruttori

Costruttore: *definizione*

- Un costruttore è un metodo speciale
- Il nome del costruttore è identico al nome della classe
- Non restituisce nessun valore
- Si può fare overloading del costruttore
- Esiste un costruttore di default che può essere eventualmente specializzato
- Viene invocato quando l'oggetto è creato

Costruttore: *esempi d'uso in C++*

```
// Pair.h - Interfaccia classe Pair
#ifndef PAIR_H
#define PAIR_H

class Pair {
  private:
     int first, second;
  public:
     Pair ();   //costruttore di default
     Pair ( int a );
     Pair ( int a, int b );
     int getFirst();
     int getSecond();
     Pair sum(const Pair & c);
};
#endif
```

```
// Main
#include "pair.h"

int main() {
  Pair x;  //x è inizializzato con il
           //costruttore di default
  …
  Pair y(); //costruttore di default
  …
  Pair z(3), w(3,-5); //altri costruttori
  …
  Pair *p = new Pair;   //costr. default
  Pair *q = new Pair(); //costr. default
  Pair *r = new Pair(3,-5);//altro costr.
};
```


Costruttori in Java

- I costruttori hanno lo stesso nome della classe
- I costruttori non hanno tipo di ritorno
- I costruttori sono richiamati dall'operatore **new** al momento della creazione di un oggetto che viene posto in memoria dinamica
- I costruttori ammettono **overloading**
- I parametri dei costruttori (e dei metodi in generale) non posso avere dei valori di default
- Però gli attributi possono essere inizializzati a valori di default e quindi i costruttori, in questo caso, non devono occuparsene.

(segue)

…Costruttori

- Un costruttore viene chiamato dopo che Java ha:
  - **Allocato** memoria per l'oggetto
  - **Inizializzato** gli attributi ai valori iniziali o ai valori di default (**0** per i numeri, **null** per gli oggetti, **false** per i boolean e **'\0'** per i caratteri)
- Se un costruttore non è definito per una classe, l'oggetto viene comunque generato quando si utilizza una **new**

distruttori

1. **this**
2. **private:** vs **public:**
- inclusione librerie

1. funzioni di utilità
2. costruttori
3. distruttori

Distruttore: *definizione e uso in C++*

- Un distruttore è un metodo speciale
- Il nome del distruttore è formato da ~ seguito dal nome della classe
- Non riceve parametri e non restituisce nessun valore
- Non si può fare overloading del distruttore
- Come per il costruttore, esiste un distruttore di default che può essere eventualmente specializzato
- Viene invocato automaticamente quando l'oggetto sta per essere distrutto
- Rilascia (dealloca) la memoria occupata dall'oggetto
- Quando sono chiamati i costruttori ed i distruttori? *(vedere esempio classe CreateAndDestroy)*
- Quando è necessario specializzare il distruttore di default? *(vedere esempio classe MyList)*

Java: Metodi conclusivi e **garbage collection**

- In Java non esiste il concetto di distruttore, ma quello di metodo conclusivo il cui nome è `finalize()`
- `finalize()`non riceve parametri e non restituisce valori
- Appena un oggetto non viene più utilizzato (es. il riferimento non è più attivo) viene marcato come non in uso. Non esiste un operatore di **delete** per rimuovere un oggetto.
- Periodicamente si controlla l'esistenza di oggetti non in uso, per poterli rimuovere e liberare la memoria utilizzata
- Il processo che esegue questo lavoro è il *garbage collector*
- Il garbage collector richiama il metodo `finalize()` immediatamente prima di rimuovere l'oggetto