```
/*
ESERCIZIO. Definire una superclasse ContoBancario e due sue sottoclassi ContoCorrente e
ContoDiRisparmio che soddisfano le seguenti specifiche:

1) Ogni ContoBancario è caratterizzato da un saldo e rende disponibili due funzionalità di
deposito e prelievo: double deposita(double) e double preleva(double) che ritornano il saldo
aggiornato dopo l'operazione di deposito/prelievo.
2) Ogni ContoCorrente è caratterizzato anche da una spesa fissa uguale per ogni ContoCorrente che
deve essere detratta dal saldo ad ogni operazione di deposito e prelievo.
3) Ogni ContoDiRisparmio deve avere un saldo non negativo e pertanto non tutti i prelievi sono
permessi; d'altra parte, le operazioni di deposito e prelievo non comportano costi aggiuntivi e
restituiscono il saldo aggiornato.
4) Si definisca inoltre una classe ContoArancio derivata da ContoDiRisparmio. La classe
ContoArancio deve avere un ContoCorrente di appoggio: quando si deposita una somma S su un
ContoArancio, S viene prelevata dal ContoCorrente di appoggio; d'altra parte, i prelievi di una
somma S da un ContoArancio vengono depositati nel ContoCorrente di appoggio.
*/

class ContoBancario {
private:
  double saldo;
public:
  double deposita(double x) {
    return x>=0 ? saldo += x : saldo;
  }

  double preleva(double x){
    return x>=0 ? saldo -= x : saldo;
  }

  double getSaldo() const {return saldo;}

  ContoBancario(double s=0.0): saldo(s>=0? s : 0) {}
};

class ContoCorrente: public ContoBancario {
private:
  static double spesaFissa;
public:
  // se x<spesaFissa, non avviene il deposito
   double deposita(double x) {
     return ContoBancario::deposita(x-spesaFissa);
   }

  double preleva(double x){
    return ContoBancario::preleva(x+spesaFissa);
  }

  ContoCorrente(double s=0.0): ContoBancario(s) {}
};
double ContoCorrente::spesaFissa = 1.0;

class ContoDiRisparmio: public ContoBancario {
public:
  // Invariante: saldo >= 0
  double preleva(double x){
    return x<=getSaldo() ? ContoBancario::preleva(x) : getSaldo();
  }
  // ContoBancario::deposita() non necessita di ridefinizione

  ContoDiRisparmio(double s=0.0): ContoBancario(s) {}
};

class ContoArancio: public ContoDiRisparmio {
private:
  // conto di appoggio deve essere modificabile
  ContoCorrente& appoggio;
  // ContoArancio e' un ContoDiRisparmio
```

```
  // Invariante: saldo >= 0
public:
  double preleva(double x) {
    if(x<=getSaldo() && 0<=x ) { appoggio.deposita(x); return ContoDiRisparmio::preleva(x); }
    return getSaldo();
  }

  double deposita(double x) {
    if(x>=0) { appoggio.preleva(x); return ContoDiRisparmio::deposita(x); }
    return getSaldo();
  }

  ContoArancio(ContoCorrente& cc,double s=0.0): ContoDiRisparmio(s), appoggio(cc) {}
};

#include<iostream>

int main() {
  ContoCorrente cc(2000);
  ContoArancio ca(cc,1500);
  ca.deposita(350); ca.preleva(400); cc.preleva(150);
  std::cout << cc.getSaldo() << std::endl; // stampa: 1897
  std::cout << ca.getSaldo() << std::endl; // stampa: 1450
}
```


REVOLUTION
IS COMING

```
void F(orario o);
...
dataora d;
F(d);
```

```
void G(const orario& o);
...
dataora d;
G(d);
```

Differenza tra F e G:

polimorfismo del parametro

```
orario::Stampa() {...}
```

stampa

12:39:24

```
dataora::Stampa() {...}
```

stampa

23/11/2020
12:39:24

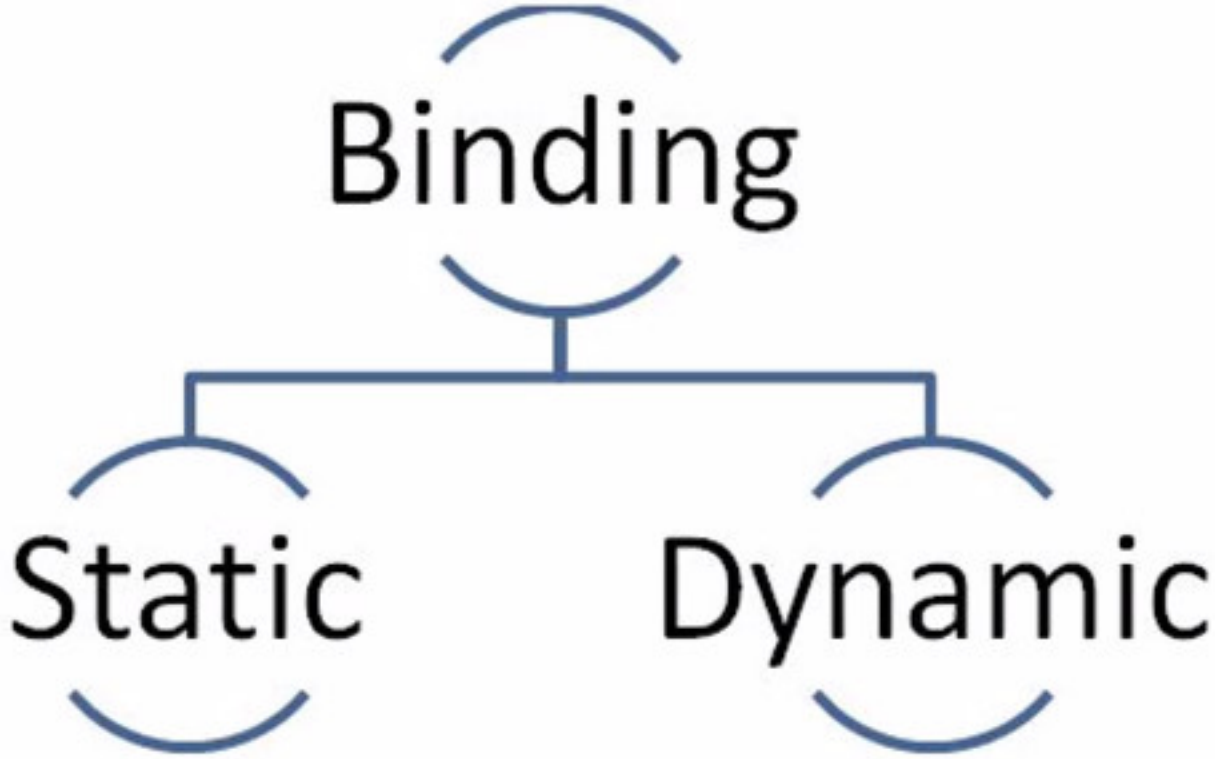
```
orario::Stampa() {...}

dataora::Stampa() {...}
```

Codice esterno:

```
void printInfo(const orario& r) { r.Stampa(); }
```

```
void printInfo(const orario* p) { p->Stampa(); }
```

Come ottenere un **comportamento polimorfo** per **G**?



Late binding
Dynamic binding
Dynamic lookup

```
class orario { ...
  virtual void Stampa(); // metodo virtuale
  ...
};

void G(const orario& o) {
  o.Stampa(); // chiamata polimorfa
}
```

```
dataora d;
orario* p = &d;
p->Stampa(); // chiamata polimorfa
```

`orario` diventa una classe **polimorfa**

`Stampa()` è un contratto **polimorfo**


Overloading
&
Overriding

Method overriding

From Wikipedia, the free encyclopedia
(Redirected from Method overriding (programming))

Method overriding, in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class.[1] The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.[2] Some languages allow a programmer to prevent a method from being overridden.

Contents [hide]

Overriding di metodi virtuali

- **identica segnatura**, tipo di ritorno e const incluso
- se la lista degli argomenti è identica ma cambia il tipo di ritorno il compilatore **segnala un errore**

Overriding di metodi virtuali

- **identica segnatura**, tipo di ritorno e const incluso
- se la lista degli argomenti è identica ma cambia il tipo di ritorno il compilatore **segnala un errore**

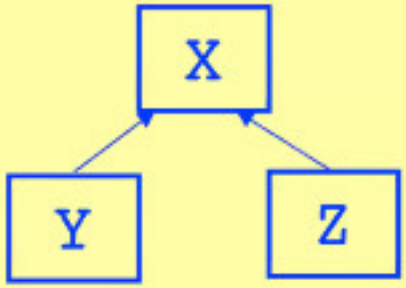
- `virtual T1* m(...)`
  `T2` $\leq$ `T1`

  overriding con tipo di ritorno ***covariante***:

  ```
  T2* m(...)
  ```

```
class B {
public:
  virtual int f() { cout << "B::f()\n"; return 1; }
  virtual void f(string s) {cout << "B::f(string)\n";}
  virtual void g() {cout << "B::g()\n";}
};
class D1 : public B {
public:
  // Overriding di un metodo virtuale non sovraccaricato
  void g() {cout << "D1::g()\n";}
};
class D2 : public B {
public:
  // Overriding di un metodo virtuale sovraccaricato
  int f() { cout << "D2::f()\n"; return 2; }
};
class D3 : public B {
public:
  // NON è possibile modificare il tipo di ritorno
  //! void f() { cout << "D3::f()\n";} // NON COMPILA
};
class D4 : public B {
public:
  // Lista degli argomenti modificata: ridefinizione e non overriding
  int f(int) { cout << "D4::f()\n"; return 4; }
};
```

```
class X {};
class Y: public X {};
class Z: public X {};

class B {
  X x;
public:
  virtual X* m() { cout << "B::m() "; return &x; }
};

class C: public B {
  Y y;
public:
  virtual X* m() { return &y; } // overriding
};

class D: public B {
  Z z;
public:
  virtual Z* m() { return &z; } // OK, overriding covariante legale
};
int main(){
  B b; C c; D d; B* pb = &b; X x;
  Y* py = c.m(); // illegale
  X* px = c.m();
  Z* pz = d.m();
  x = *(pb->m());
  pb = &d; x = *(pb->m()); // OK
}
```



Attenzione all'overriding di metodi virtuali con parametri che prevedono valori di default

```
class B {
public:
  virtual void m(int x=0) { cout << "B::m01 ";}
};

class D: public B {
public:
  // è un overriding di B::m
  virtual void m(int x) { cout << "D::m01 ";}

  // è un nuovo metodo in D e non un overriding di B::m
  virtual void m() {cout << "D::m() ";}
};

int main(){
  B* p = new D;
  p->m(2);     // stampa D::m01 e non B::m01
  p->m();      // stampa D::m01 e non D::m()
  p->B::m();   // stampa B::m01 e non D::m01
}
```

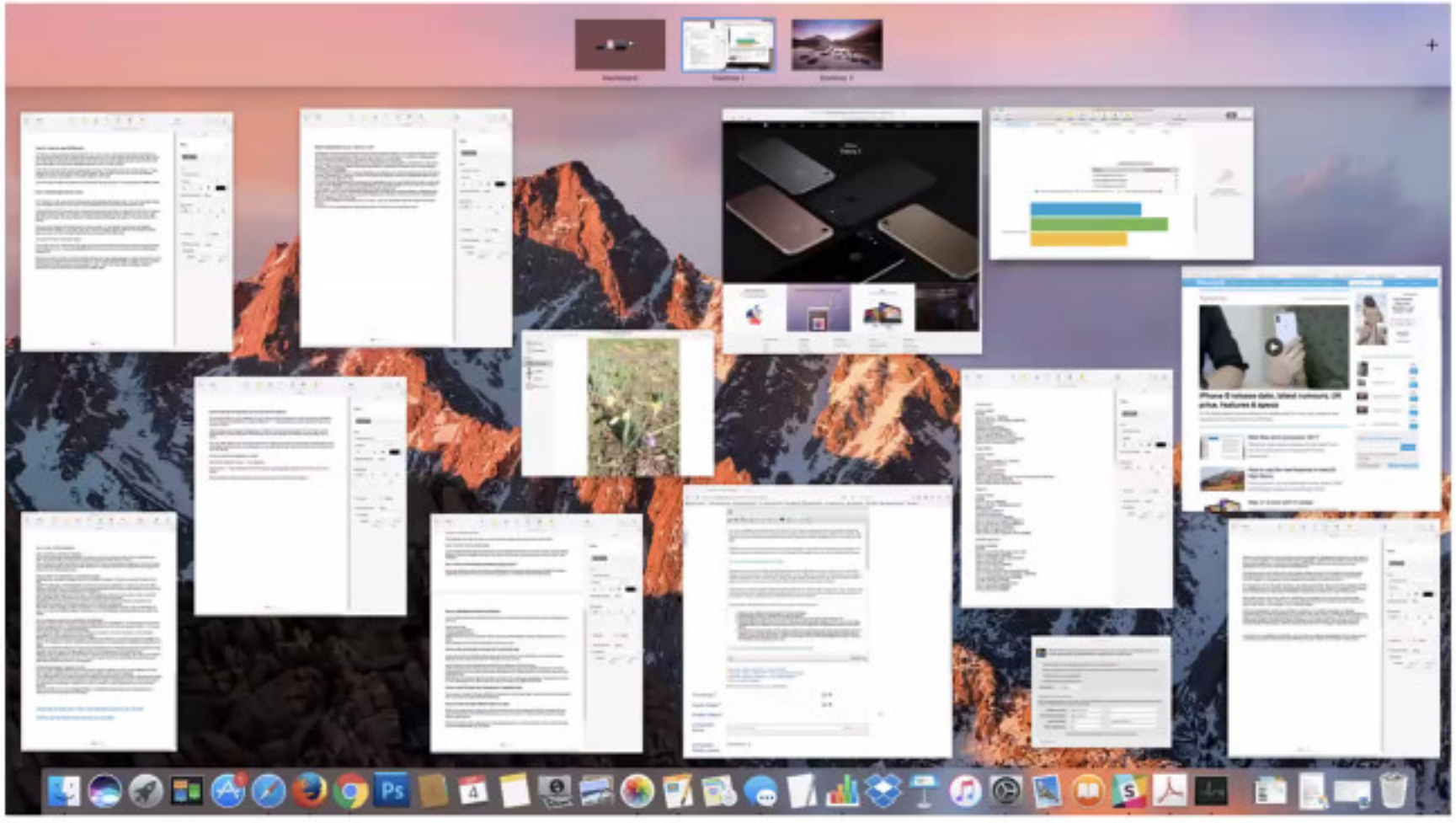
Dice il saggio

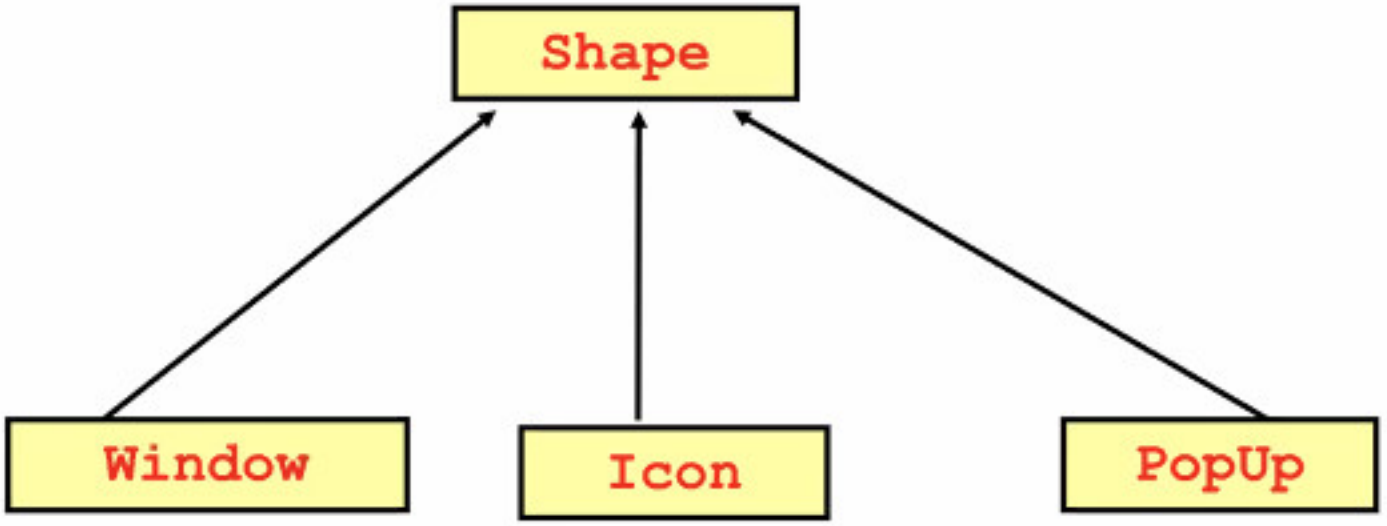
Il buon uso del polimorfismo promuove l'estensibilità del codice

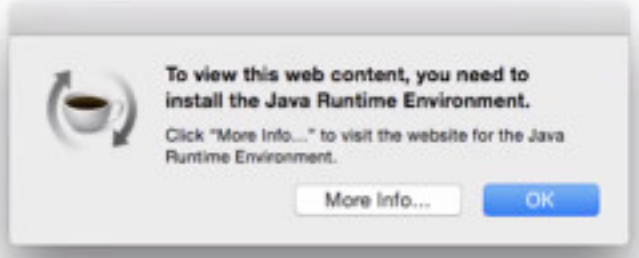
Window manager

From Wikipedia, the free encyclopedia

A **window manager** is system software that controls the placement and appearance of windows within a windowing system in a graphical user interface.[1] Most window managers are designed to help provide a desktop environment. They work in conjunction with the underlying graphical system that provides required functionality—support for graphics hardware, pointing devices, and a keyboard, and are often written and created using a widget toolkit.




Shape
Window
Icon
PopUp


To view this web content, you need to install the Java Runtime Environment.
Click "More Info..." to visit the website for the Java Runtime Environment.
More Info...
OK

```
class Shape {
   ...
   virtual void draw(Position){...}
};
class Window: public Shape {
   ...
   void draw(Position) {...}
};
class Icon: public Shape {
   ...
   void draw(Position) {...}
};
class PopUp: public Shape {
   ...
   void draw(Position) {...}
};

class DesktopManager {
   ...
   void show(const Shape& s) {
      ...
      Position p = computePosition();
      s.draw(p);
      ...
   }
};
```