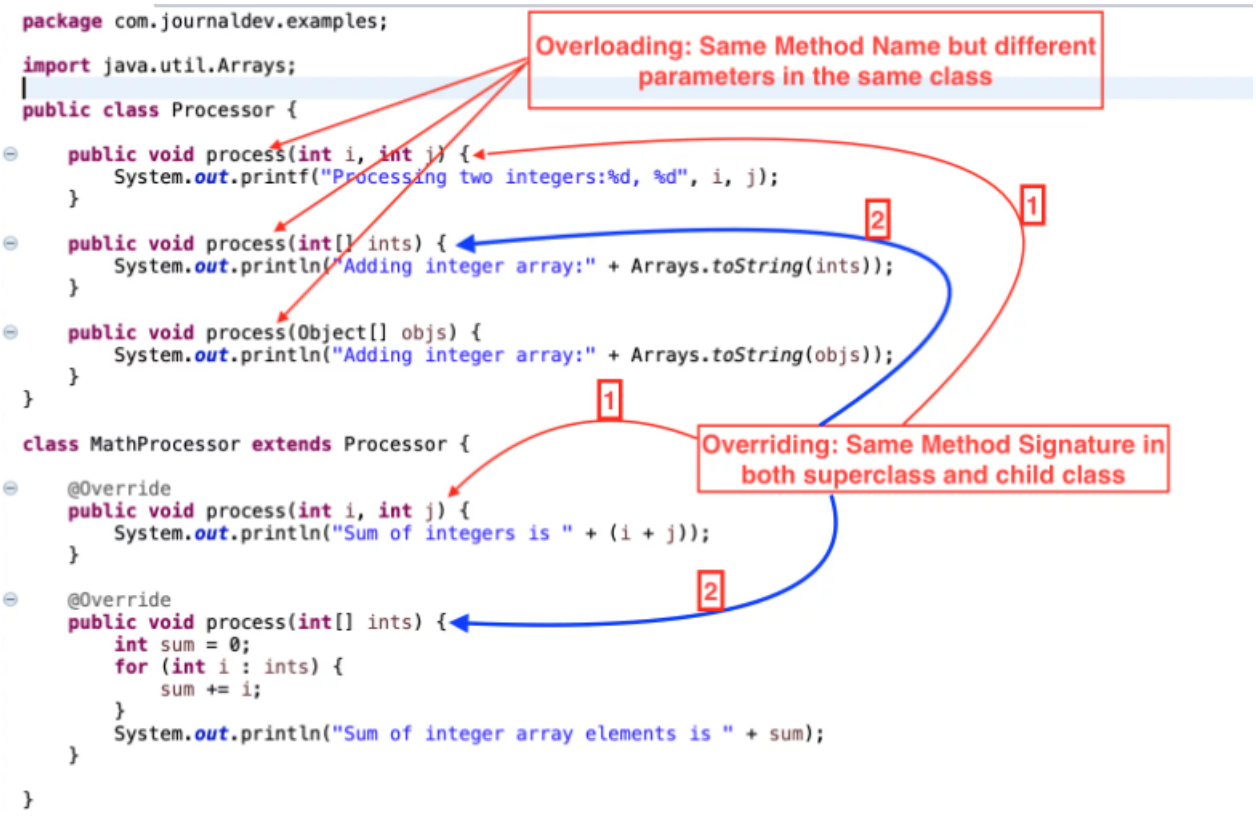
Un metodo **virtuale** è una funzione che è dichiarata con la keyword *virtual* da parte della classe base o superclasse e successivamente ridefinita da una sottoclasse.
Ciò significa che se dichiaro una funzione come *virtual* viene effettuato del polimorfismo a runtime, quindi si sceglie a seconda del contesto se chiamare la funzione della classe base oppure della classe derivata. Non è obbligatorio da parte della classe base ridefinire un semplice metodo virtuale.

La ridefinizione cambia a seconda che la funzione attuale usi lo stesso nome del metodo ma con diversi parametri nella stessa classe (**overloading**) o a seconda che ridefinisca un metodo sia nella superclasse che nella sottoclasse con la stessa segnatura (**overriding**).
L'esempio sotto è in Java, ma ben aiuta a visualizzare questo concetto.




```
package com.journaldev.examples;

import java.util.Arrays;
public class Processor {

    public void process(int i, int j) {
        System.out.printf("Processing two integers:%d, %d", i, j);
    }

    public void process(int[] ints) {
        System.out.println("Adding integer array:" + Arrays.toString(ints));
    }

    public void process(Object[] objs) {
        System.out.println("Adding integer array:" + Arrays.toString(objs));
    }
}

class MathProcessor extends Processor {

    @Override
    public void process(int i, int j) {
        System.out.println("Sum of integers is " + (i + j));
    }

    @Override
    public void process(int[] ints) {
        int sum = 0;
        for (int i : ints) {
            sum += i;
        }
        System.out.println("Sum of integer array elements is " + sum);
    }

}
```

Diventa obbligatoria la ridefinizione nel caso in cui c'è un metodo **virtuale puro o astratto**, nel qual caso una classe offre un'interfaccia (che significa semplicemente che la classe base non definisce il metodo, ma viene ridefinito da tutte le sottoclassi con quei parametri in una diversa maniera).
Alcune note:

1) Una classe è astratta se contiene almeno una funzione virtuale pura
2) Una classe diventa concreta se ridefinisce ogni metodo astratto di una classe, altrimenti rimane astratta

Ad esempio:

```
include<iostream>
using namespace std;

class Base
{
public:
    virtual void show() = 0;
};
```

```
class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}
```

Output: `In Derived`

A quel punto distinguiamo per bene i singoli tipi di binding:

- **static binding**/*early binding*/*binding statico*, avviene a compile time e vuol dire che tutte le informazioni rispetto ad una classe sono già conosciute a compile time e sa già con certezza quale tipo andare ad invocare per certo:

Ad esempio:

```
#include <iostream>
using namespace std;

class A
{
public:
    void m1()
    {
        cout << "m1 is invoked\n";
    }

    void m2()
    {
        cout << "m2 is invoked\n";
    }
};
int main()
{
    A obj;
    obj.m1();
    obj.m2();
    return 0;
}
```

- **dynamic binding**/**late binding**/*binding ritardato,* solitamente realizzata ed ottenuta tramite puntatori e/o riferimenti a runtime, che significa che a seconda del contesto di invocazione e della ridefinizione dei metodi in quell'esatto momento (quindi ritarda fino a tempo di compilazione), decide cosa è meglio chiamare. Ciascun riferimento virtuale viene mantenuto all'interno della cosiddetta *vtable*, tenendo corrispondenza di ognuno di questi.

Ad esempio:

```
#include <iostream>
using namespace std;
```

```
class B
{
public:
    // function f() is declared as virtual in the base class
    virtual void f()
    {
        cout << "The base class function is called.\n";
    }
};

class D : public B
{
public:
    void f() //function overriding
    {
        cout << "The derived class function is called.\n";
    }
};

int main()
{
    B base;    // base class object
    D derived; //derived class object

    B *basePtr = &base; // base class pointer pointing to base class
object
    basePtr->f();       //calls function of base class

    basePtr = &derived; // base class pointer pointing to object of
derived class
    basePtr->f();       //calls function of derived class

    return 0;
}
```

Nell'esempio precedente, abbiamo una classe base B in cui la funzione f() è stata dichiarata con la parola chiave "virtual". La classe derivata D eredita dalla classe base B, il che significa che tutte le funzioni membro della classe B saranno disponibili anche nella classe D. Quindi, la funzione f() della classe B sarà accessibile dalla classe D. Ma vediamo che la funzione f() è stata definita di nuovo nella classe derivata.

Ora, nel codice principale, abbiamo un puntatore di classe base *basePtr*. In primo luogo punta a un oggetto della classe "base", e la chiamata di funzione basePtr->f() chiama la funzione f() della classe base, cosa prevista anche perché il puntatore, così come l'oggetto a cui punta, sono entrambi della classe base come tipo.

Ma quando *basePtr* punta all'oggetto classe derivata "derived", è importante notare che il puntatore è di tipo classe base e l'oggetto a cui punta è di classe derivata di tipo. La funzione chiama *basePtr->f()* che viene risolto in fase di esecuzione, e quindi la funzione f() della classe derivata viene eseguita come l'oggetto è della classe derivata. Questo è un esempio di override della funzione, che è possibile a causa del late binding.

NB: Se effettuo overriding di un metodo virtuale m() in una classe D derivata da B ⇒ in D vengono nascosti tutti gli eventuali overloading di m() in B.

L'unica *eccezione* a questa regola è nel caso in cui il tipo di ritorno sia tipo puntatore o riferimento ad una classe.
Dato il metodo virtuale dove X è un tipo di classe: `virtual X* m(T1, T2, ..Tn)`

allora se Y è sottoclasse di X, è permesso che l'overriding di m() possa cambiare il tipo di ritorno in Y* (mentre la lista dei parametri deve rimanere la stessa)

Classe polimorfa = Contiene solo il distruttore virtuale
Tipo polimorfo = Avrà almeno un metodo virtuale

Costruttore nelle classi derivate: Viene sempre invocato il costruttore della classe *base* e successivamente quello della *derivata*.

1) La *derivata* costruisce il proprio sottooggetto con il costruttore della classe *base*.
2) Invoca il proprio costruttore per i campi propri
3) Viene eseguito il corpo del costruttore

Nel caso invece del costruttore di copia, anche qui:

1) Invoca il costruttore di copia della classe *base*
2) La derivata costruisce i propri campi invocando i suoi costruttori di copia

Assegnazione nelle classi derivate
L'assegnazione standard di una classe D derivata direttamente da una classe base B:
1) Invoca l'assegnazione della classe *base* (standard o ridefinita) sul sottooggetto corrispondente;
2) Successivamente esegue l'assegnazione ordinatamente membro per membro dei campi dati propri di D invocando le corrispondenti assegnazioni (standard o ridefinite);
Se l'assegnazione viene ridefinita nella derivata:
1) viene eseguito solo il suo corpo, la ridefinizione dell'assegnazione non provoca alcuna invocazione implicita

Distruttore tra classe base e derivata:
Nel caso del distruttore standard:

1) invoca il distruttore standard proprio della *derivata*
2) invoca il distruttore della classe *base*

Se invece il distruttore viene ridefinito nella derivata:

1) viene eseguito il corpo del distruttore nella *derivata*
2) viene invocato il distrttuore della classe *base* per distruggere il sottooggetto

A runtime l'invocazione dei tipi può essere determinata in due modi:

1) **typeid**, che fondamentalmente ritorna il tipo dinamico di una certa espressione altrimenti, se non contiene metodi virtuali, ritorna il tipo statico
2) **dynamic_cast**, che viene utilizzato per fare il safe downcast da una classe base ad una derivata.ì
   Quindi: B* → D* e B& → D&
   Nel caso di dynamic_cast, la classe **deve** essere polimorfa e dato TD(p) = E* : (TD = Tipo Dinamico)
   1) **Se E è sottotipo di D** ⇒ OK, **la conversione andrà a buon fine** ⇒ dynamic_cast ritornerà un puntatore di tipo D* all'oggetto obj. (effettua conversione dato che il tipo dinamico E* di p è compatibile con il tipo target D*, in quanto E* è sottotipo di D*)
   2) **Se E non è sottotipo di D** ⇒ ERRORE, **la conversione fallisce** e il dynamic_cast ritornerà un puntatore nullo. (in questo caso il tipo dinamico del puntatore non è compatibile con il tipo target)

Altra cosa importante sono i **costruttori** e **distruttori**, usati in molti esercizi di stampe. In sintesi si ricordi che:

- i costruttori vengono invocati **nello stesso ordine di creazione**
- i distruttori vengono invocati **nell'ordine inverso alla loro creazione**
- i costruttori hanno una **lista di invocazione**, per ognuno dei quali si può invocare esplicitamente il loro costruttore oppure viene chiamato di default il loro costruttore del loro tipo classe; **successivamente alla lista si esegue il corpo del costruttore**
- **Costruttore di default**: Zero parametri ed è del tipo C()
- **Costruttore di copia**: È del tipo C(C&). Esso crea un nuovo oggetto del tipo C.
- **Assegnazione profonda**: **Ridefinizione** corretta **di operator=** del tipo C& operator=(const C&); ed assegna e controlla la diversità tra riferimento costante e this, **assegnando ogni campo.**
  Ad esempio:

```
if(this != &b){ distruggi first; first=copia(b.first); } return *this;
```

- **Copia profonda**: Esso prevede C(const C&) e **copia ogni campo**.
  Ad esempio:

```
bolletta :: bolletta(const bolletta& b) : first(copia(b.first)) { }
bolletta :: copia(nodo* p){
if(!p) return 0;
nodo* primo=new nodo; primo->info=p->info; nodo* q=primo;
while(p->next){
q->next=new nodo; p=p->next; q=q->next; q->info=p->info; } q->next=0;
return primo; }
```

- **Cancellazione profonda**: Come visto finora, **una cancellazione di ogni campo** (particolare attenzione se si tratta di un oggetto sullo heap, si prevede la delete e cancellazione di tutto il resto).
  Ad esempio:

```
void bolletta::distruggi(nodo* p){
nodo* q; while(p){ q=p; p=p->next; delete q; }
```

In merito ai *contenitori list, vector*, ecc.

- Se il contenitore è **costante**, userò un **const_iterator** per scorrere gli elementi della lista
- Se il contenitore **non è costante**, userò un **iterator** invece

NB: Negli esercizi, spesso, **si va ad utilizzare l'oggetto puntato da it**. Ricordarsi sempre che si deve puntare al suo oggetto, **quindi dovrà essere correttamente dereferenziato.**
Esempio:
(*it)->getMeseCorrente(); //SI
it.getMeseCorrente(); //NO
*it.getMeseCorrente(); //NO

Tipo covariante (spiegazione di cplusplus – reference)

Se la funzione *Derived::f* esegue l'override di una funzione *Base::f*, i loro tipi restituiti devono essere uguali o essere *covarianti*. Due tipi sono covarianti se soddisfano i seguenti requisiti:

1. entrambi i tipi sono puntatori o riferimenti (lvalue o rvalue) alle classi.
2. la classe referenziata/puntata nel tipo restituito di *Base::f()* deve essere una classe base diretta o indiretta non ambigua e accessibile della classe referenziata/puntata del tipo di ritorno di *Derived::f().*

La classe nel tipo restituito di *Derived::f* deve essere Derived stessa o deve essere un tipo completo (quindi ben definito e significa che posso invocarne un oggetto perché la classe è già stata tutta definita) di *Derived::f*.

Quando viene effettuata una chiamata di funzione virtuale, il tipo restituito dal final overrider viene implicitamente convertito nel tipo restituito della funzione overridata che era stata chiamata:

```
class B {};

struct Base {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    virtual B* vf4();
    virtual B* vf5();
};

class D : private B {
    friend struct Derived; // in Derived, B is an accessible base of D
};

class A; // forward-declared class is an incomplete type

struct Derived : public Base {
    void vf1();    // virtual, overrides Base::vf1()
    void vf2(int); // non-virtual, hides Base::vf2()
//  char vf3();    // Error: overrides Base::vf3, but has different
                   // and non-covariant return type
    D* vf4();      // overrides Base::vf4() and has covariant return type
//  A* vf5();      // Error: A is incomplete type
};

int main()
{
    Derived d;
    Base& br = d;
    Derived& dr = d;

    br.vf1(); // calls Derived::vf1()
    br.vf2(); // calls Base::vf2()
//  dr.vf2(); // Error: vf2(int) hides vf2()

    B* p = br.vf4(); // calls Derived::vf4() and converts the result to B*
    D* q = dr.vf4(); // calls Derived::vf4() and does not convert
                     //  the result to B*

}
```

Esempio pratico con un tipico cosa_stampa (virtual, overriding e compagnia):

```
class A {
protected:
virtual void h() {cout<<" A::h ";}
public:
virtual void g() const {cout <<" A::g ";}
virtual void f() {cout <<" A::f "; g(); h();}
void m() {cout <<" A::m "; g(); h();}
virtual void k() {cout <<" A::k "; h(); m(); }
virtual A*n() {cout <<" A::n "; return this;}
};

class B: public A {
protected:
virtual void h() {cout <<" B::h ";}
public:
virtual void g() {cout <<" B::g ";}
void m() {cout <<" B::m "; g(); h();}
void k() {cout <<" B::k "; g(); h();}
B*n() {cout <<" B::n "; return this;}};

class C: public B {
protected:
virtual void h() const {cout <<" C::h ";}
public:
virtual void g() {cout <<" C::g ";}
void m() {cout <<" C::m "; g(); k();}
void k() const {cout <<" C::k "; h();}
};
A*p2 = new B(); A*p3 = new C(); B*p4 = new B(); B*p5 = new C(); const A*p6 = new
C();
```

Esempio di istruzioni di stampa:

```
1) p2->f();
```

p2 ha tipo statico A e tipo dinamico B. Quando effettua la chiamata alla funzione f(), essa è definita virtuale solo nella classe A; se B sottotipo la ridefinisse chiamerebbe quella, ma chiama A::f(). Si occupa poi di chiamare g(); essa è ridefinita nella sottoclasse B in g() come non costante e nella classe A come costante; andrà pertanto a stampare A::g() e poi, successivamente, chiamerà B::h(), perché ridefinito correttamente nella sottoclasse. Stampe finali: A::f A::g B::h

```
2) p2->m();
```

Osservazioni similari a prima, il metodo m() non viene ridefinito in B tipo dinamico e viene fatto il binding statico andando a chiamare A::m; quanto detto per g() e per h() vale pure qui. Stampe finali: A::m A::g B::h

```
3) p3->k();
```

p3 ha tipo statico A e tipo dinamico C. Il metodo k() viene ridefinito nella superclasse di C, quindi B e viene chiamato correttamente B::k. Il metodo g() chiama C::g() perché non si ha un A costante come riferimento in quel momento e si va nella sottoclasse C ad invocare il metodo corretto, ridefinito sempre con virtual. Successivamente, si richiede un metodo b(), che viene ridefinito dalle sottoclassi di A ma non si ha oggetto C costante al momento dell'invocazione, solo un C&. Per questa ragione si chiama il metodo h() presente nella classe B, dando luogo alla stampa B::h(). Stampa finale: B::k C::g B::h