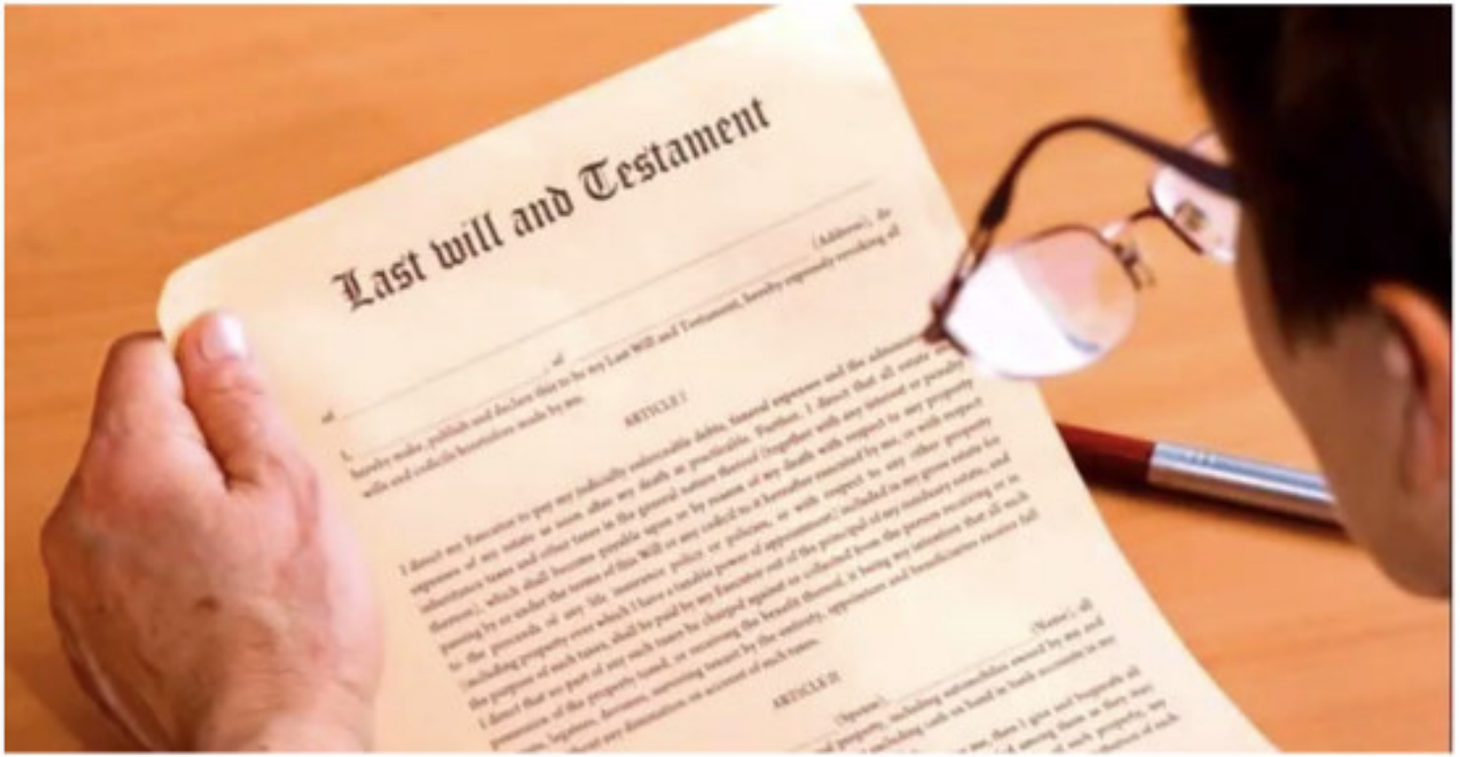
```
/*
ESERCIZIO.

Definire un template di classe dList<T> i cui oggetti rappresentano una struttura dati lista doppiamente
concatenata (doubly linked list) per elementi di uno stesso tipo T. Il template dList<T> deve soddisfare i
seguenti vincoli:

1. Gestione della memoria senza condivisione.

2. dList<T> rende disponibile un costruttore dList(unsigned int k, const T& t) che costruisce una
lista contenente k nodi ed ognuno di questi nodi memorizza una copia di t.

3. dList<T> permette l'inserimento in testa ed in coda ad una lista in tempo O(1) (cioe` costante):
-- Deve essere disponibile un metodo void insertFront(const T&) con il seguente comportamento:
dl.insertFront(t) inserisce l'elemento t in testa a dl in tempo O(1).
-- Deve essere disponibile un metodo void insertBack(const T&) con il seguente comportamento:
dl.insertBack(t) inserisce l'elemento t in coda a dl in tempo O(1).

4. dList<T> rende disponibile un opportuno overloading di operator< che implementa l'ordinamento
lessicografico (ad esempio, si ricorda che per l'ordinamento lessicografico tra stringhe abbiamo che
"campana" < "cavolo" e che "buono" < "ottimo").

5. dList<T> rende disponibile un tipo iteratore costante dList<T>::const_iterator i cui oggetti permettono
di iterare sugli elementi di una lista.
*/


template<class T>
class dList {
private:
  class nodo {
  public:
    T info;
    nodo *prev, *next;
    nodo(const T& t, nodo* p = 0, nodo* n=0): info(t), prev(p), next(n) {}
  };
  nodo *first, *last; // puntatori al primo e ultimo nodo della lista
  // lista vuota IFF first == nullptr == last

  static void destroy(nodo* n) {
    if (n != nullptr) {
      destroy(n->next);
      delete n;
    }
  }

  static void deep_copy(node *src, node*& fst, node*& last) {
    if (src) {
      fst = last = new node(src->info);
      nodo* src_sc = src->next;
        while (src_sc) {
          last = new node(src_sc->info, last);
          last->prev->next = last;
          src_sc = src_sc->next;
        }
    }
    else {
      // lista da copiare vuota
      fst = last = nullptr;
    }
  }

  static bool isLess(const nodo* l1, const nodo* l2) {
    if(!l1 && !l2) return false;
    // l1 || l2
    if(!l1) return true;// vuota < non vuota
    if(!l2) return false;// non vuota < vuota
    // l1 & l2
    if(l1->info < l2->info) return true;// T::operator<
    else if(l1->info > l2->info) return false;
    else // l1->info == l2->info
      return isLess(l1->next, l2->next);
  }
```

```
public:

  dList(const dList& l) {
    deep_copy(l.first,first,last);
  }

  dList& operator=(const dList& l) {
    if(this != &l) {
      destroy(first);
      deep_copy(l.first,first,last);
    }
    return *this;
  }

  ~dList() {destroy(first);}

  // duale a insertBack: Homework
  void insertFront(const T& t);

  void insertBack(const T& t) {
    if(last){ // lista non vuota
      last = new nodo(t,last,nullptr);
      (last->prev)->next=last;
    }
    else // lista vuota
      first=last=new nodo(t);
  }

  dList(unsigned int k, const T& t): first(nullptr), last(nullptr) {
    for(unsigned int j=0; j<k; ++j) insertFront(t);
  }

  bool operator<(const dList& l) const {
    if(this == &l) return false; // optimization: l < l e' sempre false
    return isLess(first, l->first);
  }

  const_iterator begin() const {
    return const_iterator(first);
  }

  const_iterator end() const {
    if(!last) return const_iterator();
    return const_iterator(last+1,true); // attenzione: NON e' past the end
  }

  class const_iterator {
    friend dList <T>;
  private: // const_iterator indefinito: ptr==nullptr & past_the_end==false
    const nodo* ptr;
    bool past_the_end;
    // convertitore "privato" nodo* => const_iterator
    const_iterator(nodo* p, bool pte = false): ptr(p), past_the_end(pte) {}
  public:
    const_iterator(): ptr(nullptr), past_the_end(false) {}
    const_iterator& operator++();
    const_iterator operator++(int); // postfisso
    const_iterator& operator--(); // prefisso
    const_iterator operator--(int); // postfisso
    bool operator==(const const_iterator&) const;
    bool operator!=(const const_iterator&) const;
    const T& operator*() const; // perche' e' un const_iterator
    const T* operator->() const; // perche' e' un const_iterator
  };

};
```



EREDITARIETA'

Rapporto di parentela in linea retta

```
// dichiarazione classe orario
class orario {
  friend ostream& operator<<(ostream&,const orario&);
public:
  orario(int o = 0, int m = 0, int s = 0);
  int Ore() const;
  int Minuti() const;
  int Secondi() const;
  orario operator+(const orario&) const;
  bool operator==(const orario&) const;
  bool operator<(const orario&) const;
private:
  int sec;
};
```


MON TUE WED THU FRI SAT SUN
JAN FEB MAR APR MAY JUN
JUL AUG SEP OCT NOV DEC
23 59
DEC 25 WED
0 1 2 3 4 5 6 7 8 9

ADT **dataora**, una data con orario: **31/12/1999 ore 23:59**

31/12/1999 ore 23:59

In particolare, un valore dataora **è** (anche) **un** orario

eredito da

```
class dataora : public orario {



};
```

eredito da

```
class dataora : public orario {
private:
   int giorno;
   int mese;
   int anno;
public:
   int Giorno() const;
   int Mese() const;
   int Anno() const;
};
```

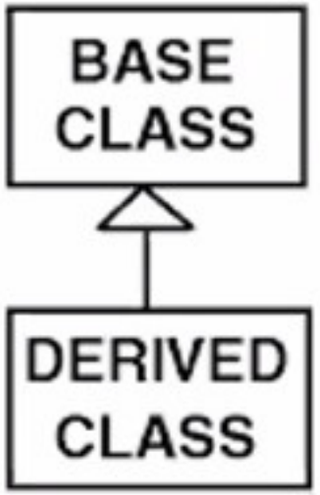
e aggiungo membri

Terminologia

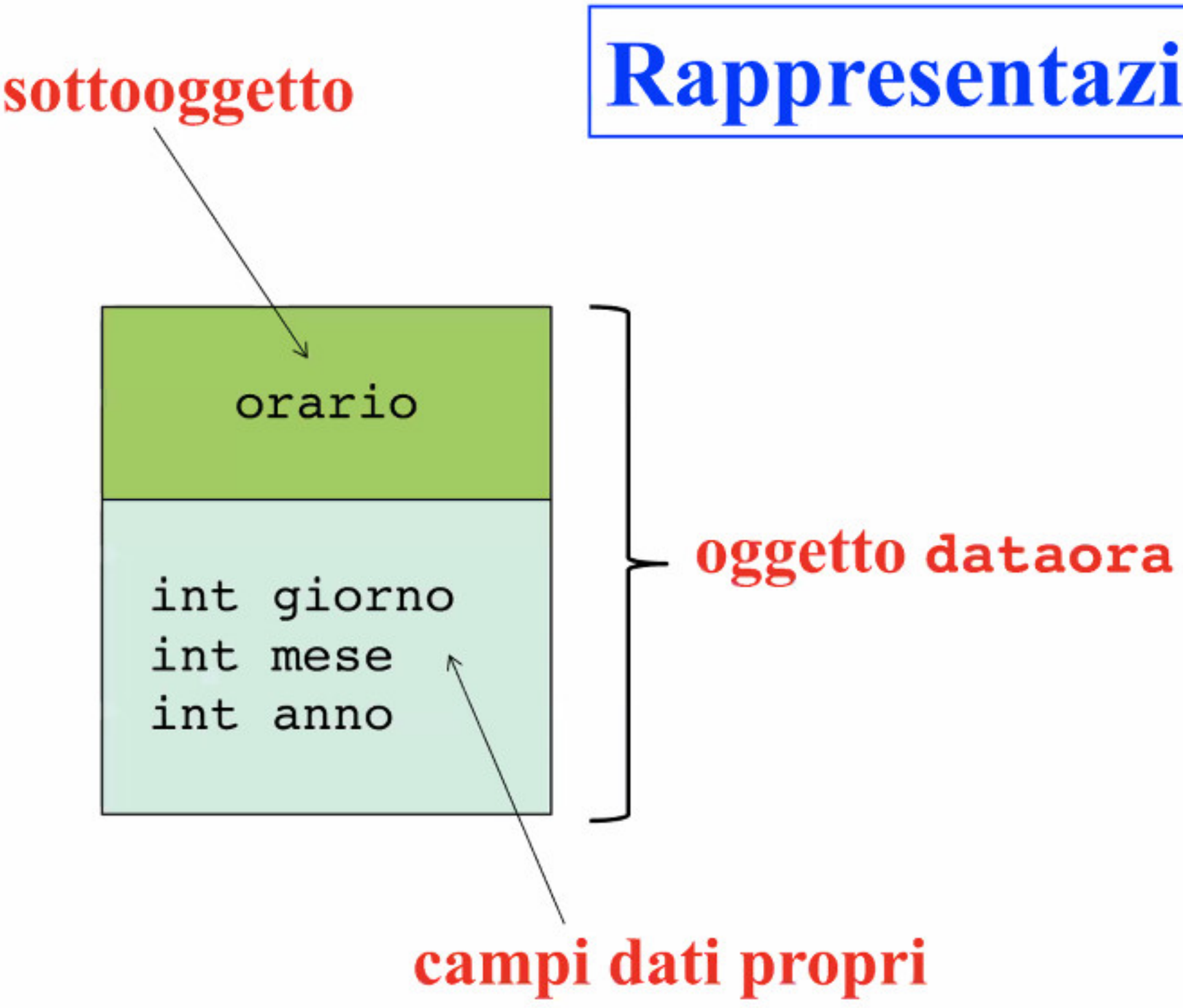
Classe base **B** e classe derivata **D**

Sottoclasse **D** e superclasse **B**

Sottotipo **D** e supertipo **B**



Rappresentazione



Relazione “is-a” induce il **subtyping** (o subsumption) che è la **caratteristica fondamentale dell’ereditarietà**:

Subtyping

From Wikipedia, the free encyclopedia

In programming language theory, **subtyping** (also **subtype polymorphism** or **inclusion polymorphism**) is a form of type polymorphism in which a **subtype** is a datatype that is related to another datatype (the **supertype**) by some notion of substitutability, meaning that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype. If S is a subtype of T, the subtyping relation is often written S <: T, to mean that any term of type S can be *safely used in a context where* a term of type T is expected. The precise semantics of subtyping crucially depends on the particulars of what "safely used in a context where" means in a given programming language. The type system of a programming language essentially defines its own subtyping relation, which may well be trivial.

Polymorphism

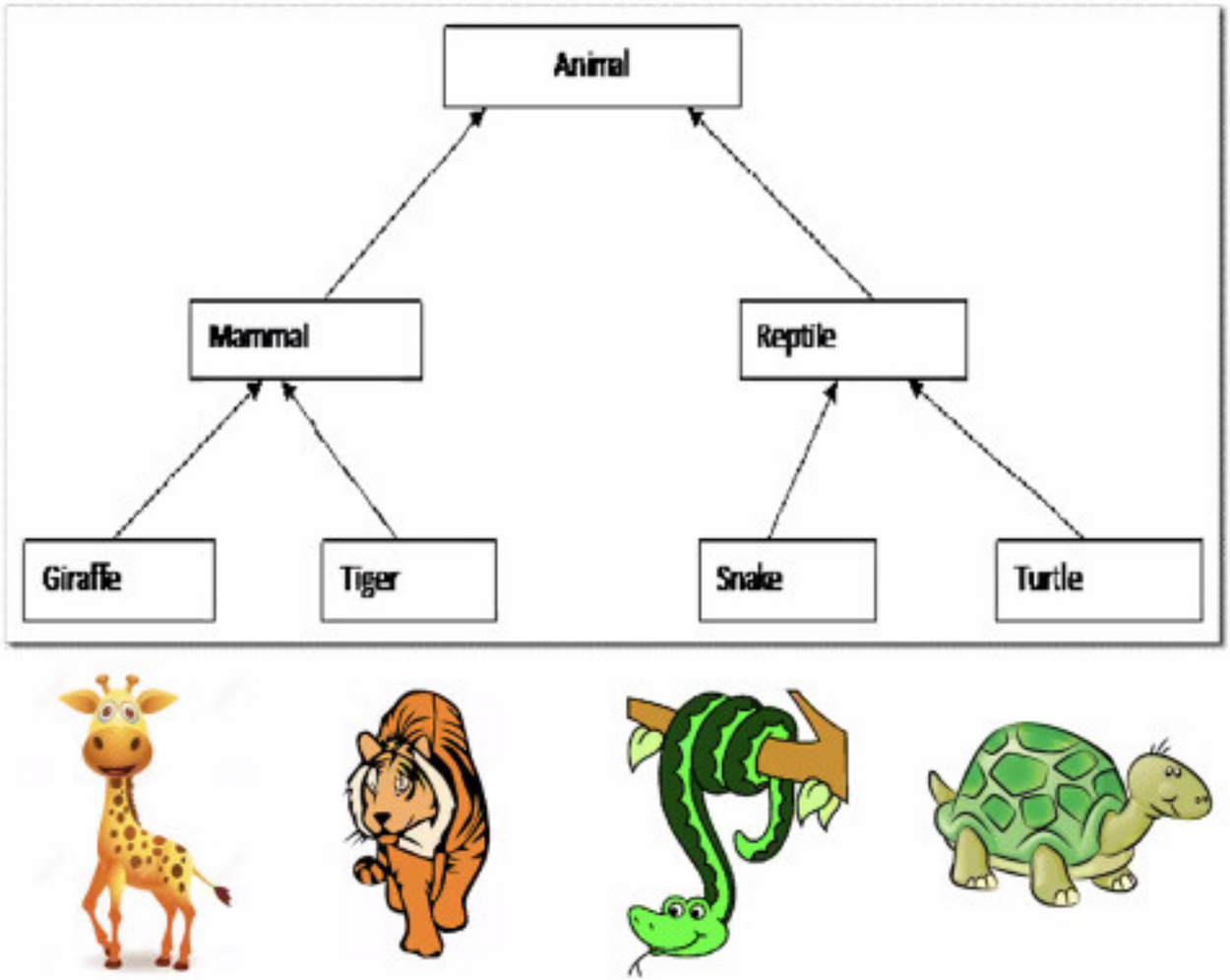
Ad hoc polymorphism
Function overloading
Operator overloading
Parametric polymorphism
Double dispatch
Multiple dispatch
Single & dynamic dispatch
Subtyping
Virtual function

V • T • E

Due to the subtyping relation, a term may belong to more than one type. Subtyping is therefore a form of type polymorphism. In object-oriented programming the term 'polymorphism' is commonly used to refer solely to this *subtype polymorphism*, while the techniques of parametric polymorphism would be considered *generic programming*.

Relazione “is-a” induce il **subtyping** (o subsumption) che è la **caratteristica fondamentale dell’ereditarietà**:

Ogni oggetto della classe derivata è utilizzabile anche come oggetto della classe base



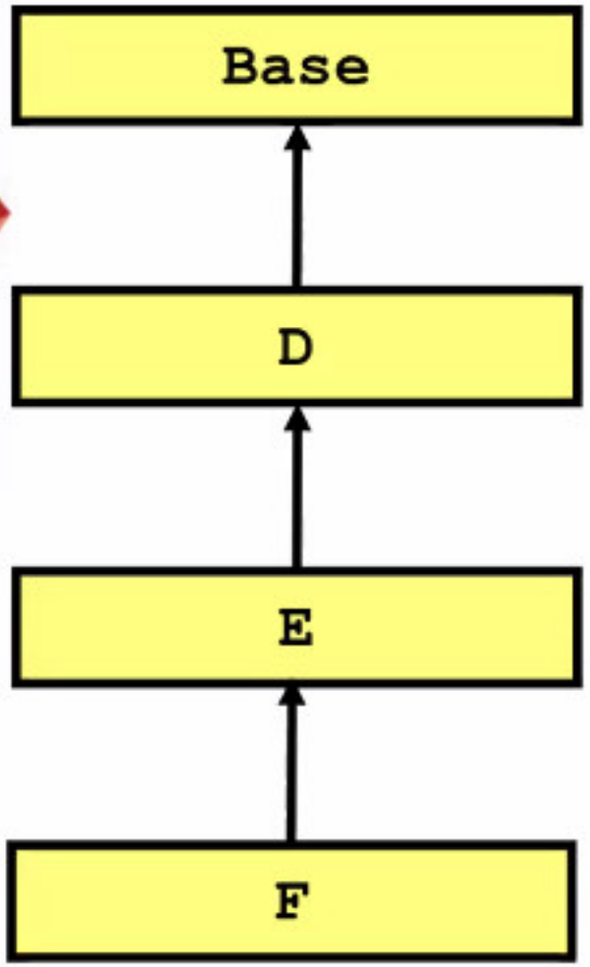
- **Subtyping**: Sottotipo **`D`** $\Rightarrow$ Supertipo **`B`**
- Per oggetti: **`D`** $\Rightarrow$ **`B`** **estrae il sottooggetto**

Subtyping per puntatori e riferimenti

$$D^* \Rightarrow B^* \qquad D\& \Rightarrow B\&$$

Puntatori e riferimenti polimorfi

Gerarchie di classi: sottotipi diretti ed indiretti



Casí d'uso di ereditarieta

1) Estensione
2) Specializzazone
3) Ridefinizione
4) Riutilizzo di codice

Ereditarietà per **estensione**

```
dataora <: orario
```

Ereditarietà per **specializzazione**

`QButton <: QComponent`

Ereditarietà per **ridefinizione**

```
Queue <: List
```

Ereditarietà per **riuso di codice** **non è subtyping**

`Queue reuse List`