Genericità

- Definizione
- Realizzazione in C++ e in Java

Genericità: *esempio introduttivo*

- Supponiamo di dover gestire la lista dei contatti telefonici personali
- Supponiamo che, a tale scopo, decidiamo di progettare e realizzare una classe **ElencoTelefonico**.
- La classe **ElencoTelefonico** offre servizi per effettuare le classiche operazioni di gestione: *ricerca, inserimento, cancellazione, modifica*
- Un oggetto **ElencoTelefonico** contiene oggetti di tipo **Utente**
- La classe **ElencoTelefonico** è implementata utilizzando strutture dati complicate (per gli attributi) ed algoritmi molto efficienti (per le operazioni)

Genericità: *ancora sull'esempio*

- Se fosse necessario dover gestire (stesse operazioni) anche la lista dei **CD** e dei **libri** personali si potrebbe recuperare il codice prodotto per la classe **`ElencoTelefonico`**:
  - copiare la classe **`ElencoTelefonico`** nella classe **`ElencoCD`**, sostituendo le occorrenze di oggetti **Utente** con oggetti di tipo **`CD`**
  - copiare la classe **`ElencoTelefonico`** nella classe **`ElencoLibri`**, sostituendo le occorrenze di oggetti **`Utente`** con oggetti di tipo **`Libro`**
- Questo **riuso** di codice può aumentare notevolmente la produttività ma richiede il mantenimento di tre copie di codice quasi identico !!

Genericità: *definizione*

- In un caso come questo, la genericità fornisce un notevole aiuto.
- Se definiamo **`Elenco`** come classe **parametrica** (in C++ le classi parametriche sono note come **template**), significa che almeno una delle classi utilizzate all'interno di **`Elenco`** non deve necessariamente essere assegnata fino al momento della compilazione/esecuzione.
- Questa presumibilmente sarebbe la classe degli elementi da gestire nel particolare tipo di elenco che istanziamo.

La **genericità** è la costruzione di una classe C in modo tale che una o più delle classi che essa utilizza internamente venga fornita solo in fase di compilazione/esecuzione

Genericità: esempi

Es:

- Una classe `Elenco` definita indipendentemente dalla definizione degli elementi che fanno parte dell'elenco.
- I contenitori della STL.
- I contenitori in java.util.* di Java
- Una classe `Coppia` (che vedremo in seguito)

Genericità: *uso*

Usando la genericità è pertanto possibile progettare la classe `Elenco` scrivendo:

```
class Elenco< ClasseElementoElenco >{
...
var elemento: ClasseElementoElenco :=
  ClasseElementoElenco.New();
...
elemento.print()
... }
```

Attenzione!!

Una volta disponibile la classe parametrica `Elenco`, le specifiche classi sono derivate istanziando l'oggetto interno:

```
var elencoTelefono: Elenco := Elenco.New( <Telefono> );
var elencoCD: Elenco := Elenco.New( <CD> );
var elencoLibri: Elenco := Elenco.New( <Libro> );
...
```

Template: classi parametriche

- Le classi **template** sono utilizzate quando almeno una classe dei sui membri non è nota.
- Come si realizzano in C++ le classi parametriche?
- Iniziamo con un esempio semplice: la classe `Coppia`.

La classe `Coppia`: *esempio introduttivo*

- Supponiamo di dover gestire delle coppie di elementi in diversi ambiti applicativi.
- Supponiamo che, a tale scopo, decidiamo di progettare e realizzare una classe **`Coppia`**
- La classe **`Coppia`** offre semplici metodi per restituire gli elementi (primo, secondo) e per fare dei confronti (operatori < e ==) tra coppie.
- Un oggetto **`Coppia`** contiene due oggetti di tipo non specificato.

La classe “coppia”

```
template <class T1, class T2>
class Coppia {
 private:
    T1 primo;              //primo elemento della coppia
    T2 secondo;            //secondo elemento della coppia
 public:
  Coppia() : primo(), secondo(){}  //costruttore di default e con parametri
  Coppia(const T1& a, const T2& b) : primo(a), secondo(b){}

  void setPrimo(const T1& a){ primo = a;}        //alcuni metodi
  void setSecondo (const T2& b){ secondo = b;}
  T1 getPrimo() const {return primo;}
  T2 getSecondo() const {return secondo;}
                                                 //gli operatori
  bool operator==(const Coppia<T1, T2>& x) const {
    return primo == x.primo && secondo == x.secondo; }

  bool operator<(const Coppia<T1, T2>& x) const {
    return primo < x.primo || (primo == x.primo && secondo <x.secondo);}
};
```


Classi Template: *limiti e uso in C++*

- Non è possibile dichiarare e definire i metodi di una classe template in file separati (es: coppia.h e coppia.cpp) poiché il compilatore non può compilare finché non sono definiti i tipi parametri della classe.
- Un oggetto di una classe template va dichiarato specificando sempre i tipi parametrici della classe.

```
#include ''coppia.h''
...
Coppia<int, bool> x(1,true);
...
Coppia<Prof, Scuola> * p= new Coppia<Prof, Scuola>();
...
```

Template: funzioni

- E' possibile definire funzioni template. es:

```
template <class T>
   T max (T a,  T b) { if (a > b) return a; else return b;};
```

- Al posto di …

```
int max (int a,  int b) { … };
double max(double a, double b) { … };
string max(string a, string b) { … };
:
:
.
```

Stesso codice !!

La classe "Elenco" in Java - definizione

```
public class Elenco<T> {

 private LinkedList<T> l=new LinkedList<T>(); //attributo

 public Elenco() { }                          //costruttore

 public void push(T x) {  l.addFirst(x); }

 public T top(){return l.getFirst(); }

 public void pop(){l.removeFirst(); }

 public void print(){
   Iterator<T> iterator = l.iterator();
   while(iterator.hasNext())
      System.out.print(iterator.next().toString());}
}
```

La classe “Elenco” in Java - uso

```
public class Client {
        ...
        public static void main(String[] args) {
         ...
         provac.Elenco<String> e;
         e = new provac.Elenco<String>();
         e.push("Uno");
         e.push("Due");
         e.push("Tre");
         ...
         e.print();
         ...
    }

}
```