Programmazione ad Oggetti Mod.2

Nicola Panizzolo

February 2023

Contents

Chapter 1

Il linguaggio Java

Java è un linguaggio **class-centrico**, non puoi scrivere funzioni senza definire classi.
Fondamentale il **polimorfismo**, come per tutti gli OOPL, senza di esso non ha senso la programmazione a oggetti.

Durante il corso le classi verranno scritte nello stesso file(**nested**), per comodità, in un programma serio le classi andrebbero implementate in file separati.

1.1 Lezione 1

1.1.1 Statement ed Espressioni

1. Uno **statement** è un pezzo di codice che dichiara e inizializza qualcosa, quindi esegue un **Binding**
   Esempi di statement sono return, while, if, else,...

```
int i = x + 8
Animal fido = new Dog(50, "Red")
```

2. Una **espressione** è un pezzo di codice che calcola qualcosa.
   Esempio di espressione

```
new Dog(50, "Red")
```

1.1.2 Subsuction

Il compilatore java verifica se il tipo di sinistra è **più specializzato** del tipo di destra, e in caso positivo dà errore, quando si lega un'espressione a una variabile c'è un solo verso.
I **cast** sono forzature date dal programmatore.

```
Animal fido = new Animal(6);
Dog ciccio = new Dog(6,"nero");

fido = ciccio //NON DA' ERRORE

ciccio = fido //DA' ERRORE!!

ciccio = (Dog)fido //Forzatura del programmatore
```

1.1.3 Method Dispatching(sacro miracolo)

```
Dog pluto = new Dog(20,"Dotted");
Animal ciccio = pluto;

ciccio.eat(pluto) //chiama il metodo eat della classe Dog()
```

Grazie al **Method Dispatching** ciccio chiamerà il metodo eat() all'interno della classe **Dog**, e non della classe Animal. Il Method Dispatching è la vera **essenza del polimorfismo**,

1.2 Lezione 2

Dichiarare variabili **non vuol dire** costruire oggetti, gli oggetti si costruiscono con la keword **new**.

```
1 ciccio.eat(new Dog(10, "white"));
```

In questo codice di esempio non ho dichiarato nessuna varibaile, ma ho creato un nuovo oggetto.

1.2.1 Subtyping

Il **polimorfismo** consente un maggiore riutilizzo del codice. Un algoritmo che prende in input un array di int non può funzionare con un array di float, andrebbe riscritto e ciò non favorisce il riuso del codice. Per questo si utilizzano i **generics**, ma senza scomodarli la stessa cosa può essere fatta utilizzando un array di **Object** col **subtyping**, che è l'unione di Subsuction e Method Dispatching.

I generics sono stati aggiunti in java nel 1999 con la versione 1.5, mentre java è nato nel 1994, questo implica che fino al 1999 si è programmato senza utilizzare i generics.

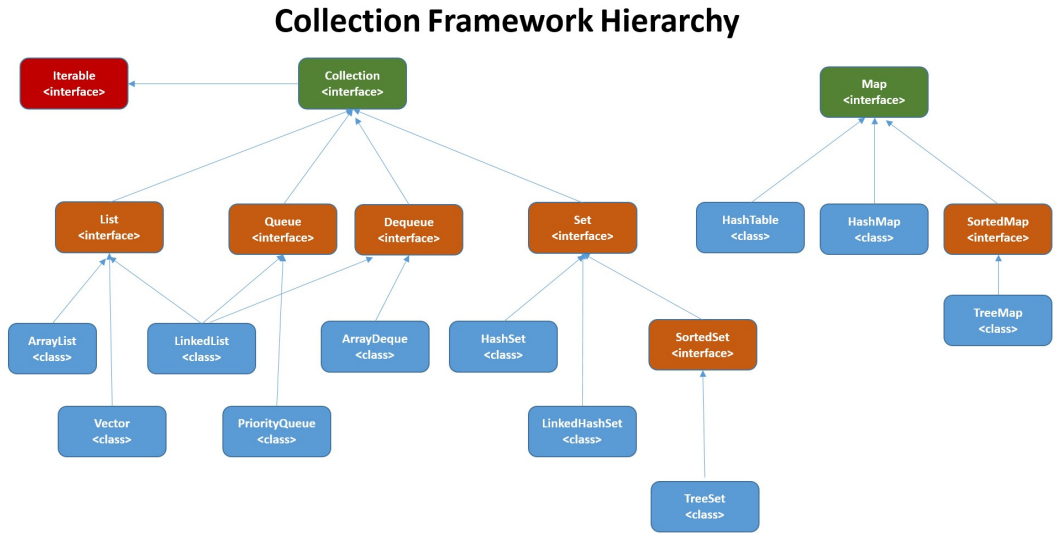

Figura 1.2.1

Implementazione senza generics

Senza utilizzare i generics per avere il polimorfismo devo utilizzare sempre la classe Object.

```
1  public class Generics{
2      public interface Iterator{
3          boolean hasNext();
4          Object next(); //non ci sono i generics nel 1994
5      }
6      //JDK 1.0 (1994)
7      public interface Iterable{
8          Iterator iterator();
9      }
10
11     public interface Collection extends Iterable{
12         void add(Object x);
13         void remove(Object x);//x puo' essere un indice, un valore, qualsiasi cosa
14     }
```

```
15
16      public interface List extends Collection{
17          Object get(int index);
18          void set(int index, Object x);
19      }
20  }
```

Un **Iterable** è un oggetto che **puoi iterare**, cioè che ha un fottuto iteratore(Iterator).

List è un oggetto **indicizzabile**, aggiunge a Collection getter e setter.

Dire che metodo **equals** della classe Object permette di effettuare un confronto strutturale è una **grandissima minchiata**, se volete confrontare gli oggetti dovete farvi voi **quel cazzo di equals**.

1.3 Lezione 3

Ecco un esempio di implementazione di una struttura dati senza utilizzare i generics.

```
1   public class ArrayList implements Generics.List
2   {
3           private Object[] a;
4           private int size;
5
6           public ArrayList()
7           {
8               this.a = new Object[10];
9               this.size = 0;
10          }
11
12          //Capacity
13          public ArrayList(int capacity)
14          {
15                  this.a = new Object[capacity];
16              this.size = capacity;
17          }
18
19          @Override
20          public int size(){ return this.size; }
21
22          public Generics.Iterator iterator() { return null; }
23
24          @Override
25          public void add(Object x)
26          {
27                  if(this.size >= a.length)
28              {
29                  Object[] old = this.a;
30                  this.a = new Object[old.length * 2];
31                  for(int i = 0; i < old.length; ++i)
32                          a[i] = old[i];
33              }
34              a[this.size++] = x;
35          }
36
37          @Override
38          public void remove(Object x) {}
39
40          @Override
41          public Object get(int index) { return this.a[index]; }
42
43
44          @Override
45          public void set(int index, Object x) { this.a[index] = x; }
46
47          @Override
48          public boolean equals (Object o) { return super.equals(obj); }
49  }
```

Osservazioni :

1. **Orribile**: Non sono mai sicuro al 100% su che tipo c'è dentro ad ogni cella, era responsabilità del programmatore effettuare il **downcasting**;
2. **Insicuro**: Il programmatore deve ricordare tutto, la cosa si risolve rendendo più "intelligenti" i tipi, quindi utilizzando i generics;
3. Presenza di **bugs**;
4. Presenza di **runtime errors**.

1.3.1 Utilizzo reale 'no generics'

```
1  public static void main(String[] args)
2  {
3      List c = new ArrayList();
4      //Uso collection per essere il piu' generico possibile
5      //"Ad oggetti si programma ai minimi"
6      //"Il tipo piu' alto possibile che ha i metodi che ti servono"
7      //La subsumption minima va usata per riusare il codice, se domani
8      //mi sveglio e creo LinkedList sempre da Collection, devo solo cambiare una riga.
9      //Queste add sono obrobri
10     //Aggiungo cose a caso di tipi diversi
11     //Il compilatore e' scemo? No, gli stai dando Object e per lui va tutto bene.
12     // Per essere polimorfo dovevo usare Object, pero' cosi' sono troppo polimorfo.
13     //"Ora subsumo un pelo meno" mi serve get().
14     c.add("Nome");
15     c.add(8);
16     c.add(new Dog(25, "Brown"));
17     for(int i = 0; i < c.size(); ++i)
18         {
19         Object s = c.get(i); // Un folle dice string e spano' lo
20         //brucia minacciandolo di mandarlo in terza elementare.
21         String s = (String) c.get(i);
22         //Il compilatore e' nel chilling, gia' all'indice i = 1 RuntimeError.
23         // e' come mettere il pilota manuale ma tu non sai guidare.
24         }
25 }
```

1.3.2 Esempio con generics

```
// JDK 1.5 (2004)
public class Generics
{
        // Lo strumento con cui iteri
        public interface Iterator<T>
    {
        boolean hasNext();
        T next();
    }
        // Puoi scorrere
        public interface Iterable<T>
    {
        Iterator<T> iterator();
    }

        // Puoi aggiungere e togliere elementi
        public interface Collection<T> extends Iterable<T>
    {
        void add(T o);
        void remove(T o);
        boolean contains(T o);
    }

        // List -> Si può accedere tramite indice
        public interface List<T> extends Collection<T>
    {
        T get(int index);
        void set(int index, T x);
    }
}

public class ArrayList<T> implements Generics.List<T>
{
        private T[] a;
        private int size;
        public ArrayList()
        {
            this.a = new T[10];
            this.size = 0;
        }

        //Capacity
        public ArrayList(int capacity)
        {
            this.a = new T[capacity];
            this.size = capacity;
        }

        @Override
        public int size(){ return this.size; }

        //Qui e' un type parameter
        public Generics.Iterator<T> iterator() { return null; }

        @Override
        public void add(T x)
        {
                if(this.size >= a.length)
            {
                T[] old = this.a;
                this.a = new T[old.length * 2];
                for(int i = 0; i < old.length; ++i)
                        a[i] = old[i];
            }
            a[this.size++] = x;
        }

        @Override
```

```
69          public void remove(T x) {}
70
71          @Override
72          public T get(int index) { return this.a[index]; }
73
74          @Override
75          public void set(int index, T x) { this.a[index] = x; }
76
77          @Override
78          public boolean equals (Object o) { return super.equals(obj); }
79  }
```

1.4 Lezione 4 - Type parameter vs argument

Parameter è attaccato alla classe, **Argument** va dopo(es. nella new).
esempio:

```
1  public interface Myinterface<A,B,C>
```

In questo caso <A,B,C> sono **parameter**, perché è la prima volta che li nomino.

```
1  public interface MyInterface<A, B, C>
2  extends Iterable<A>, Iterator<Collection<B>>
3  {
4      A metodo(B x) ; //non si scrive public perche' e' sottointeso
5      void metodo2(C c);
6  }
```

In questo caso sto passando il type **parameter** A di riga 1 come **argument** di Iterable.

Esempio di implementazione di questa interfaccia:

```
1  public static class MyInterfaceImplementation
2  implements MyInterface<Integer, String, List<Boolean>>
3  //Integer = A, String = B, List<Boolean> = c
4  {
5      public Integer m(String x) {}
6      public void p(List<Boolean> c) {}
7
8      //metodi di Iterable
9      public Iterator<Integer> iterator(){}
10
11     //metodi di Iterator
12     public boolean hasNext(){}
13     public Collection<String> next(){}
14 }
```

Devo implementare tutti i metodi di MyInterface, e tutti i metodi delle classi che MyInterface estende(Iterator e Iterable).

Parentesi Array

Java implementa un vero e proprio Tipo Array, al contrario di C, in cui si tratta solo di un puntatore. Quando inzializzo un array in Java **non** viene chiamato il costruttore, perchè crea sempre **array vuoti**, alloca solo lo spazio necessario.

```
1  //in Java
2      int[] a = new int[56];
3  //in C
4      int *b;
```

Come si può vedere da questo esempio in Java esiste un vero e proprio tipo *int*[].

Java non lascia istanziare Collection che hanno come tipo un Type Argument

```
List<String>[] b new List<String>[400]; //NON SI PUO' FARE
```

1.4.1 Type Erasure

La **type erasure** è presente solo in Java, non è implementata da altri linguaggi. Come è stato detto prima di Java 5 non erano presenti i Generics, il cambiamento così radicale creò non pochi problemi a tutti le aziende che usavano Java. Molte di queste aziende si occupavano di transazioni, e richiedevano un software che durasse per molti anni, infatti vollero che Java avesse la **retrocompatibilità**, fondamentale per mantenere i vecchi software. Quando i programmatori di Java implementarono i Generics dovettero "scompilare" il linguaggio, perchè i nuovi programmi Java dovevano funzionare anche sulle JVM più vecchie, pre-generics.

```
public static class ArrayList<T> implements List<T> {
```

Il **bytecode** generato da questa riga di codice è **senza Generics**, il compilatore supporta i Generics, ma vengono poi trasformati. Questo stratagemma ha risolto il problema della retrocompabilità che però ha un prezzo, il runtime non è in grado di discriminare tra tipi si differenziano tra loro solo per un Type Argument. Esempio:

```
//ERRORE DEL COMPILATORE
public static class MyClass implements Iterable<String>, Iterable<Integer> {
```

Il compilatore darà un errore perché una volta compilato il codice dovrà **togliere** i Type Argument, questa pratica si chiama **Type Erasure**. Risulterebbe una cosa del genere:

```
public static class MyClass implements Iterable, Iterable {
//ovviamente non ha senso implementare 2 volte la stessa interfaccia Iterable
```

Type Erasure ha inoltre un problema di **overloading**, non si può fare un overloading di un metodo differenziandosi dal primo di un Type Argument, perché il compilatore non è in grado di distinguere i Type Argument a causa del Type Erasure.

```
1 public static class MyCLass implements Iterable, Iterable {
2     public void f (String s) {)
3     public void f (Integer n) {}
4     public void f(List<String> U){}//ERRORE
5     public void f(List<Double> L){}
6 }
```

In questo esempio i metodi f di riga 4 e 5 risultano uguali al compilatore.

1.4.2 Static Iterator

In questo esempio siamo all'interno della Classe ArrayList, e stiamo implementando un iteratore per ArrayList.

```
public static class ArrayList<T> implements List<T> {
[...]
    private class MyIterator implements Iterator<T> {
        private int pos = 0;

        @Override
        public boolean hasNext() {
            return pos < size();
        }

        @Override
        public T next() {
            return get(pos++);
        }
    }

    private static class MyStaticIterator<E> implements Iterator<E> {
        private int pos = 0;
        private ArrayList<E> l;

        public MyStaticIterator(ArrayList<E> l) {
            this.l = l;
        }
```

```
24
25          @Override
26          public boolean hasNext() {
27              return pos < l.size();
28          }
29
30          @Override
31          public E next() {
32              return l.get(pos++);
33          }
34      }
35
36      @Override
37      public Iterator<T> iterator() {
38          return null;
39          //return new MyIterator();
40          return new MyStaticIterator<>(this);
41          // TODO usare una anonymous class
42      }
43      //vado ad iterare
44      public static void main() {
45          List<Integer> l = new ArrayList<Integer>();
46          Iterator<Integer> it = l.iterator();
47          while (it.hasNext()) {
48              Integer n = it.next();
49              System.out.println(n);
50          }
51      }
52  [...]
53  }
```

Parentesi static Un **metodo static** non può riferirsi agli altri membri della sua classe, un **campo static** non può riferirsi agli altri membri della sua classe. Static non permette di vedere i propri fratelli. Nell'esempio soprastante, siccome non posso accedere ai membri della classe ArrayList, mi faccio passare un ArrayList<E> all'interno del costruttore.

1.5 Lezione 5

1.6 Lezione 6

1.6.1 Funzione di hashing

La funzione di hashing prende un **qualsiasi elemento**(albero, vettore, stringa, ...) e la "trasforma" in un numero. Una funzione di hashing restituisce lo stesso numero se utilizzata su due elementi uguali. L'obiettivo di una funzione di hashing non è trasformare in un numero e poi tornare indietro, l'obiettivo è quello di **effettuare confronti**.

Ogni interfaccia del JDK ha una propria funzione di Hashing.

1.6.2 HashSet

L'interfaccia Set utilizza le **funzioni di hashing** per fare i confronti, assumendo che l'utilizzatore di Set si sia fatto il suo metodo di hashing.

In questa implementazione utilizzeremo l'ArrayList creato in precedenza(privato), così facendo avremo accesso ai metodi di ArrayList e non dovremo impazzire per la riallocazione della memoria con gli array Java nativi.

```
1  public class HashSet<T> implements Set<T>{
2      private List<T> l = new ArrayList<>();
3      //Java mi permette di inizializzare senza creare un costruttore di default
4      //normalmente avremmo scritto solo private List<T> l;
```

```
    //e poi lo avremmo inizializzato nel costruttore
    @Override
    public int size(){
        return l.size();
    }

    @Override
    public void add(T x){
    //dynamic dispatching, utilizzo il mio metodo contains al posto di l.contains()
    //perche' se qualcuno vuole fare l'Override di contains non deve modificare add()
        if(!contains(x))
            l.add(x)
    }

    //qui non utilizzo la remove di arrayList perche' devo usare Hash
    @Override
    public void remove(T x){
        for(int i = 0; i < l.size(); ++i){
            T o = l.get(i);
            if(o.hashCode() == x.hashCode())
                l.removeAt(i);
        }
    }




    @Override
    public boolean contains(T x){
        for(int i = 0; i < l.size(); ++i){
            T o = l.get(i);
            if(o.hashCode() == x.hashCode())
                return true;
        }
        return false;
    }

    @Override
    public Iterator<T> iterator(){
        return l.iterator();
    }

}
```

Per la funzione **remove** ci serve un removeAt() su List<>, quindi abbiamo riscritto List con l'aggiunta della removeAt().

Java è un mondo di cose che si possono spaccare a Runtime per motivazioni strane.

1.6.3 Queue

```
public interface Queue<T> extends Collection<T>{
    void push(T x);
    T pop();
    T peek();
}
```

La nostra classe che implememenenta Queue<T> si chiamerà BasicQueue.

```
public class BasicQueue<T> implements Queue<T>{
    private List<T> l = new ArrayList<>();

    @Override
    public int size(){return l.size()};

    @Override
    public void add(T x){
        l.add(x);
    }
```

```
11
12      @Override
13      public void remove(T x){
14          throw new NotImplementedException();
15      }
16
17      @Override
18      public boolean contains(T x) {
19          return false;
20      }
21
22
23      @Override
24      public Iterator<T> iterator() {
25          //la new ha funzioni diverse in base a cosa e' seguito
26          return new Iterator<T>(){  //Anonymous class
27              private int pos = 0;
28
29              @Override
30              public boolean hasNext(){ return pos < size(); }
31
32              @Override
33              public T next() { return get(pos++); }
34          }
35      }
36
37      @Override
38      public void push(T x) {
39
40      }
41
42      @Override
43      public T pop() {
44          return null;
45      }
46
47      @Override
48      public T peek() {
49          return null;
50      }
51
52  }
```

1.7 Lezione 7

1.7.1 Mappe

Una **mappa** è una struttura dati polimorfa che associa a ogni valore una chiave.

Interfaccia

```java
public class Pair<A, B> {
    private final A first;
    private final B second;

    public Pair(A f, B s){
        this.first = f;
        this.second = s;
    }
}
```

```java
public interface Map<K, V> extends Iterable<Pair<K, V>>{

    //non scrivo 'static' perche' una classe nested in un interfaccia si comporta
    //come static
    class KeyNotFoundException extends Exception {
        public KeyNotFoundException(String message) {
            super(message);
        }
    }

    //la keyword 'put' da piu' l'idea di una mischia di elementi,
    //per questo non usiamo 'add'
    void put(K key, V value);

    //prende una chiave e restituisce il suo valore corrispondente
    V get(K key) throws KeyNotFoundException;

    boolean containsKey(K key);

    //ritorna il valore che va a togliere
    V remove(K key) throws KeyNotFoundException;
}
```

1.7.2 Implementazione HashMap

```
1  public class HashMap<K, V> implements Map<K, V>{
2      private static int CAPACITY = 1000000;
3      private List<List<Pair<K,V>>> l = new ArrayList<>(CAPACITY);
4
5      @Override
6      public Iterator<Pair<K, V>> iterator() {
7          //TODO fixare e riscrivere per casa
8          return new Iterator<Pair<K, V>>() {
9              Iterator<List<Pair<K,V>>> it = l.iterator();
10             private int i = 0, j = 0;
11             private boolean _hasNext = true;
12
13             @Override
14             public boolean hasNext() {
15                 return i + 1 < l.size();
16             }
17
18             @Override
19             public Pair<K, V> next(){
20                 Pair<K,V> r = null;
21                 if(j >= 0){
22                     List<Pair<K,V>> inl = l.get(i);
23                     r = inl.get(j++);
24                     if(j > inl.size())
25                         j = -1;
26                 }else {
27
28                     for (; i < l.size(); ++i) {
29                         List<Pair<K, V>> inl = l.get(i);
30                         if (inl == null)
31                             continue;
32                         r = inl.get(0);
33                         j = 1;
34                     }
35                     if(r == null)
36                         _hasNext = false;
37                     return r;
38                 }
39             }
40         };
41     }
42
43     @Override
44     public void put(K key, V value) {
45         int h = key.hashCode() % CAPACITY;
46         List<Pair<K,V>> inl = l.get(h);
47         if(inl == null){
48             inl = new ArrayList<>();
49             l.set(h, inl);
50         }
51         inl.add(new Pair<>(key,value));
52     }
53 h
54     @Override
55     public V get(K key) throws KeyNotFoundException {
56         int h = key.hashCode() % CAPACITY;
57         List<Pair<K, V>> inl = l.get(h);
58
59     }
60
61     @Override
62     public boolean containsKey(K key) {
63         return false;
64     }
65
66     @Override
67     public V removeKey(K key) throws KeyNotFoundException {
68         return null;
```

```
    }

    @Override
    public V remove(K key) throws KeyNotFoundException {
        return null;
    }
}
```

1.8 Lambdas

```
    public class Lambdas {

        /*interface Function<A, B> {
            B apply(A x);
        }*/

        /*interface Consumer<T> {
            void accept(T x);
        }*/

        /*interface Supplier<T> {
            T get();
        }*/

        /*interface Runnable {
            void run();
        }*/

        public static void main(String[] args) {

            Runnable r = () -> { System.out.println("ciao"); };
            Runnable r2 = new Runnable() {
                @Override
                public void run() {
                    System.out.println("ciao");
                }
            };

            Supplier<Integer> h = () -> 1;
            Supplier<Integer> h2 = new Supplier<Integer>() {
                @Override
                public Integer get() {
                    return 1;
                }
            };

            Consumer<Integer> g = x -> { System.out.println(x); };
            Consumer<Integer> g2 = new Consumer<Integer>() {
                @Override
                public void accept(Integer x) {
                    System.out.println(x);
                }
            };

            Function<Integer, Integer> f = x -> x + 1;
            Function<Integer, Integer> f2 = new Function<Integer, Integer>() {
                @Override
                public Integer apply(Integer x) {
                    return x + 1;
                }
            };

        }

        public static <A, B> Collection<B> map(Collection<A> c,
                                               Function<A, B> f) {
            Collection<B> r = new ArrayList<>();
```

```
58              for (A x : c) {
59                  r.add(f.apply(x));
60              }
61              return r;
62          }
63
64          public static <A, B> Iterator<B> mapIterator(Iterator<A> it,
65                                                       Function<A, B> f) {
66              return new Iterator<B>() {
67                  @Override
68                  public boolean hasNext() {
69                      return it.hasNext();
70                  }
71
72                  @Override
73                  public B next() {
74                      return f.apply(it.next());
75                  }
76              };
77          }
78
79          public static <T> Collection<T> filter__pure(Collection<T> c,
80                                                       Predicate<T> p) {
81              Collection<T> r = new ArrayList<>();
82              for (T x : c) {
83                  if (p.test(x))
84                      r.add(x);
85              }
86              return r;
87          }
88
89          public static <T> void filter__impure(Collection<T> c,
90                                                Predicate<T> p) {
91              Iterator<T> it = c.iterator();
92              while (it.hasNext()) {
93                  if (!p.test(it.next()))
94                      it.remove();
95              }
96          }
97
98          public static <T> void iter(Collection<T> c, Consumer<T> f) {
99              for (T x : c) {
100                 f.accept(x);
101             }
102         }
103
104         public static <T> T sum(Collection<T> c, T zero, BiFunction<T, T, T> f) {
105             return foldLeft(c, zero, f);
106         }
107
108         public static <A, B> B foldLeft(Collection<A> c,
109                                         B zero,
110                                         BiFunction<A, B, B> f) {
111             B z = zero;
112             for (A x : c) {
113                 z = f.apply(x, z);
114             }
115             return z;
116         }
117
118
119         public static void main2(String[] args) {
120             List<String> l = new ArrayList<>();
121             l.add("ciao");
122             l.add("mi");
123             l.add("chiamo");
124             l.add("pippo");
125             String s = sum(l, "", (x, y) -> x + y);
126
127         }
128     }
```

1.9 wildcard

I wildcard sono la feature più ad alto livello in assoluto nel linguaggio Java.

```
1  public class Wildcards {
2
3      //funzione identità coi generics
4      public static <T> T identity(T x){
5          return x;
6      }
7      //prendo in input un tipo T e restituisco un tipo T
8
9
10
11     //funzione identità col subtyping
12     public static Object identity2(Object x){
13         return x;
14     }
15     //in questo caso posso ricevere in input una stringa e restitutire una lista, SBAGLIATISSIMO
16
17
18
19     public static Collection<?> identity3(Collection<?> x){ //equivalente a una Collection di OBJECT
20         return x;
21     }
```

1.10 Covarianza, controvarianza e invarianza

Nei moderni linguaggi di programmazione orientati agli oggetti tipo Java, C#, C++ e così via, che offrono il supporto sia del polimorfismo per inclusione (espresso dall'ereditarietà) sia del polimorfismo parametrico (espresso dai generici), esiste un problema importante legato all'assenza, di default, di una "sinergia comportamentale" tra i citati sistemi polimorfi.

In sostanza, quanto detto significa che, per il polimorfismo per inclusione, data una classe Base e una classe Derived da essa derivata, sarà sempre lecito assegnare un'istanza di tipo Derived in una variabile di tipo Base e ciò perché esiste sempre una relazione di sottotipo Derived <: Base (il simbolo <: denota tale relazione).

Nel contempo, per il polimorfismo parametrico, data una classe generica Generic<T> e le classi parametrizzate Generic<Base> e Generic<Derived> non sarà mai lecito assegnare un'istanza di tipo Generic<Derived> in una variabile di tipo Generic<Base> e ciò perché non esiste mai una relazione di sottotipo Generic<Derived> <: Generic<Base>.

Ciò detto, appare evidente che in un linguaggio di programmazione orientato agli oggetti, moderno e ben progettato, debba essere presente un meccanismo che consenta di esprimere una relazione "sinergica" di sottotipo, detta varianza, tra i tipi complessi (per esempio classi generiche e dunque proprie del polimorfismo parametrico) e i tipi utilizzati per costruirli (per esempio i tipi forniti per gli argomenti di tipo e dunque propri del polimorfismo per inclusione). Date, quindi, le classi Generic<T>, Base e Derived, potremmo avere i seguenti casi:

1. **Covarianza**: La relazione di sottotipo espressa da Generic<Base> e Generic<Derived> è preservata rispetto a quella espressa da Base e Derived, ossia è lecito e type safe assegnare un'istanza di tipo Generic<Derived> in una variabile di tipo Generic<Base>. In modo più formale, data una classe generica C<T>, essa è definita **covariante** rispetto a T se la relazione di sottotipo tra le classi D <: B implica la stessa relazione di sottotipo tra le classi C<D> <: C<B>.

2. **Controvarianza**: La relazione di sottotipo espressa da Generic<Base> e Generic<Derived> è invertita rispetto a quella espressa da Base e Derived, ossia è lecito e type safe assegnare un'istanza di tipo Generic<Base> in una variabile di tipo Generic<Derived>. In modo più formale, data una classe generica C<T>, essa è definita **controvariante** rispetto a T se la relazione di sottotipo tra le classi D <: B implica una relazione di sottotipo tra le classi C<B> <: C<D>.

3. **Invarianza**: La relazione di sottotipo espressa da Generic<Base> e Generic<Derived> è ignorata rispetto a quella espressa da Base e Derived, ossia non è lecito e type safe produrre degli assegnamenti tra un tipo Generic<Base> e un tipo Generic<Derived>. In modo più formale, data una classe generica C<T>, essa è definita **invariante** rispetto a T solo quando la relazione di sottotipo tra le classi C<D> <: C<B> è valida per D = B.

In definitiva l'espressione di una varianza consente di variare tra dei tipi complessi, per esempio dei tipi generici, la loro relazione di sottotipo in modo covariante (va nella stessa direzione) o controvariante (va in direzione opposta) rispetto alla relazione di sottotipo dei loro tipi costituenti, per esempio gli argomenti di tipo forniti per la costruzione di un tipo effettivo. Se non c'è alcuna variazione, allora i tipi complessi rimarranno invarianti, ossia non varieranno la loro relazione di sottotipo rispetto a quella dei loro tipi costituenti.

Per quanto attiene al linguaggio Java, di default tutti i tipi generici sono **invarianti**, ma è comunque possibile esprimere sugli argomenti di tipo la loro varianza, ossia definire se essi sono covarianti o controvarianti.

Snippet errato

```
1   abstract class Dog { }
2
3   class WhiteTerrier extends Dog { }
4   class GoldenRetriever extends Dog { }
5
6   public class Snippet_9_7
7   {
```

```
8         public static void main(String[] args)
9         {
10            List<WhiteTerrier> wt = new ArrayList<>();
11            List<Dog> dogs = new ArrayList<>();
12
13            dogs = wt; // error: incompatible types:
14                       // List<WhiteTerrier> cannot be converted to List<Dog>
15
16            // se fosse lecito l'assegnamento dogs = wt sarebbe possibile mettere a
17            // compile time e a runtime un oggetto di tipo WhiteTerrier in una lista di
18            // WhiteTerrier
19            dogs.add(new WhiteTerrier());
20
21            // se il compilatore permettesse di aggiungere un WhiteTerrier a una lista di Dogs
22            // allora andrebbe in errore aggiungendo un Golden Retriever
23            dogs.add(new GoldenRetriever());
24        }
25    }
```

1.10.1 Annotazioni di varianza e wildcard

Java consente di "aggirare" la limitazione descritta precedentemente di mancanza di relazione di ereditarietà tra i tipi generici (invarianza) consentendo di specificare, durante l'utilizzo di un tipo generico, per ogni argomento di tipo, una cosiddetta **annotazione di varianza**, ossia un'annotazione che indicherà se il relativo parametro di tipo sarà covariante oppure controvariante.

Annotazione di covarianza

```
1     // Annotazione di covarianza: wildcard upper bound.
2
3     <? extends bound_type>
```

In pratica, dato un tipo T, <? extends T> indica che sarà possibile utilizzare qualsiasi sottotipo di T oppure T stesso. La sintassi di covarianza permette dunque di esplicitare un'annotazione di covarianza con cui, in altre parole, dato un tipo List<S> potremo sempre legittimamente assegnare un oggetto del suo tipo a una variabile di tipo List<? extends T> se S è di tipo T o un sottotipo di T.

annotazione di controvarianza

```
1     // Annotazione di controvarianza: wildcard lower bound.
2
3     <? super bound_type>
```

In pratica, dato un tipo T, <? super T> indica che sarà possibile utilizzare qualsiasi supertipo di T oppure T stesso. La sintassi di controvarianza permette dunque di esplicitare un'annotazione di controvarianza con cui, in altre parole, dato un tipo List<S> potremo sempre legittimamente assegnare un oggetto del suo tipo a una variabile di tipo List<? super T> se S è di tipo T o un supertipo di T.

1.11 Esercizi

1.11.1 BST - esame 1 Giugno 2023

```java
public static class BST<T> implements Iterable<T> {
    protected final Comparator<? super T> cmp;
    protected Node root;
    protected class Node {
        protected final T data;
        protected Node left, right;
        protected Node(T data, Node left, Node right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }

    public BST(Comparator<? super T> cmp) {
        this.cmp = cmp;
    }

    public void insert(T x) {
        root = insertRec(root, x);
    }

    protected Node insertRec(Node n, T x) {
        /* da implementare */

        if(n == null){
            return new Node(x, null, null);
        }

        if(cmp.compare(x , n.data) > 0)
            return insertRec(n.left, x);
        else if(cmp.compare(x , n.data) < 0)
            return insertRec(n.right, x);

        return n;
    }

    protected void dfsInOrder(Node n, Collection<T> out) {
        /* da implementare */
        if(n != null){
            dfsInOrder(n.left, out);
            out.add(n.data);
            dfsInOrder(n.right, out);
        }
    }

    @Override
    public Iterator<T> iterator() {
        /* da implementare */
        ArrayList<T> dfs = new ArrayList<>();
        dfsInOrder(root,dfs);
        return dfs.iterator();
    }

    public T min() {
        /* da implementare */
        if(root == null) return null;
        Node tmp = root;
        while(tmp.left != null){
            tmp = tmp.left;
        }
        return tmp.data;
    }


```

```
65      public T max() {
66          /* da implementare */
67          if(root == null) return null;
68          Node tmp = root;
69          while(tmp.right != null){
70              tmp = tmp.right;
71          }
72          return tmp.data;
73      }
74
75  }
76
77  public static class ComparableBST<T extends Comparable<? super T>> extends BST<T>{
78      ComparableBST(){
79          super(new Comparator<T>() {
80              @Override
81              public int compare(T o1, T o2) {
82                  return o1.compareTo(o2);
83              }
84          });
85
86          /*VARIANTI
87
88          super( Comparator::compareTo );
89
90          super( (o1, o2) -> o1.compareTo(o2) );
91
92          */
93      }
94  }
```

1.11.2 classe Treenode - esame del 13/9/2022

```
    public static void main(String[] args) {
        TreeNode<Integer> t1 =
                TreeNode.lr(1,
                        TreeNode.lr(2,
                                TreeNode.v(3),
                                TreeNode.v(4)),
                        TreeNode.r(5,
                                TreeNode.lr(6,
                                        TreeNode.v(7),
                                        TreeNode.v(8))));

        TreeNode<Integer> t2 =
                TreeNode.lr(1,
                        TreeNode.r(5,
                                TreeNode.lr(6,
                                        TreeNode.v(7),
                                        TreeNode.v(8))),
                        TreeNode.lr(2,
                                TreeNode.v(3),
                                TreeNode.v(4)));


        // test pretty printer
        System.out.println("pretty printer:");
        System.out.println("t1: " + t1);
        System.out.println("t2: " + t2);

        // test equality
        System.out.print("equality: ");
        System.out.println(t1.equals(t2) + ", " + (t1.left != null ? t1.left.equals(t2.right) : ""));

        // test iterator
        System.out.print("iterator: ");
        for (Integer n : t1) {
            System.out.printf("%d ", n);
        }
        System.out.println();

    }

    public static class TreeNode<T> implements Iterable<T> {

        @NotNull
        private final T data;
        @Nullable
        private final TreeNode<T> left, right;
        @Nullable
        private TreeNode<T> parent = null;

        // 1.c

        public TreeNode(@NotNull T data, @Nullable TreeNode<T> left, @Nullable TreeNode<T> right) {
            this.data = data;
            this.left = left;
            this.right = right;
            if (left != null) left.parent = this;
            if (right != null) right.parent = this;
        }

        // i seguenti pseudo-costruttori aiutano a costruire alberi in modo più succinto e
        //controllato rispetto all'innestamento dei costruttori

        // solo ramo sinistro
        public static <T> TreeNode<T> l(@NotNull T data, @NotNull TreeNode<T> left) {
            return new TreeNode<>(data, left, null);
        }

```

```
69          // solo ramo destro
70          public static <T> TreeNode<T> r(@NotNull T data, @NotNull TreeNode<T> right) {
71              return new TreeNode<>(data, null, right);
72          }
73
74          // ramo sinistro e destro
75          public static <T> TreeNode<T> lr(@NotNull T data,
76                                           @NotNull TreeNode<T> left,
77                                           @NotNull TreeNode<T> right) {
78              return new TreeNode<>(data, left, right);
79          }
80
81          // foglia
82          public static <T> TreeNode<T> v(@NotNull T data) {
83              return new TreeNode<>(data, null, null);
84          }
85
86          // 1.b
87
88          @Override
89          public boolean equals(@Nullable Object o) {
90              if (o instanceof TreeNode) {
91                  BlockingQueue<Integer> b;
92
93                  TreeNode<T> t = (TreeNode<T>) o;
94                  return areEqual(data, t.data) && areEqual(left, t.left) && areEqual(right, t.right);
95              }
96              return false;
97          }
98
99          private static boolean areEqual(@Nullable Object a, @Nullable Object b) {
100             return a == b || (a != null && a.equals(b));
101             //return Objects.equals(a, b);
102             // alternativamente si può usare questo metodo del JDK
103         }
104
105         // 1.e
106
107         @Override
108         @NotNull
109         public String toString() {
110             return String.format("%s%s%s", data, left != null ? String.format("(%s)", left) : "", right != nu
111                     String.format("[%s]", right) : "");
112         }
113
114         // 1.a
115
116         @Override
117         public Iterator<T> iterator() {
118             return iterator_easy();
119             // stub ad una delle due implementazione; cambiare lo stub per testare l'altra
120         }
121
122
123         // questo è l'implementazione facile,
124         //suggerita pubblicamente dal docente in classe duranto l'appello del 13/9/22
125         private Iterator<T> iterator_easy() {
126             Collection<T> r = new ArrayList<>();
127             dfs(r);
128             return r.iterator();
129         }
130
131         private void dfs(Collection<T> c) {
132             c.add(data);
133             if (left != null) left.dfs(c);
134             if (right != null) right.dfs(c);
135         }
136
137
138
139
```

```
140
141         // questo è l'implementazione ottimizzata, che attraversa l'albero senza liste d'appoggio
142         private Iterator<T> iterator_opt() {
143             return new Iterator<>() {
144                 @Nullable
145                 private TreeNode<T> current = TreeNode.this;
146
147                 @Override
148                 public boolean hasNext() {
149                     return current != null;
150                 }
151
152                 @Nullable
153                 private static <T> TreeNode<T> getNextNode(@NotNull TreeNode<T> n) {
154                     if (n.left != null)
155                         return n.left;
156                     else if (n.right != null)
157                         return n.right;
158                     else {
159                         while (n.parent != null) {
160                             final TreeNode<T> last = n;
161                             n = n.parent;
162                             if (n.right != null && n.right != last)
163                                 return n.right;
164                         }
165                         return null;
166                     }
167                 }
168
169                 @Override
170                 @NotNull
171                 public T next() {
172                     T r = current.data;
173                     current = getNextNode(current);
174                     return r;
175                 }
176             };
177         }
178
179     }
```

1.11.3 Sequenze contigue di Fibonacci - esame 1/7/2022

Si implementi in Java 8+ una classe FiboSequence le cui istanze rappresentano sequenze contigue di numeri di Fibonacci di lunghezza data in costruzione. Tali istanze devono essere iterabili tramite il costrutto foreach di Java, devono pertanto implementare l'interfaccia parametrica del JDK java.util.Iterable<T>.

```
//(a) 6 punti:

public static class FiboSequence implements Iterable<Integer>{
    private final int len;

    protected int fib( int n){ //protected mi permette di overridare
        if(n < 2){
            return 1;
        }else{
            return fib(n-1) + fib(n-2);
        }
    }

    FiboSequence( Integer len ){
        this.len = len;
    }


    @Override
    public Iterator<Integer> iterator(){
        return new Iterator<>() {
            private int i = 0;
            @Override
            public boolean hasNext() {
                return i < len;
            }

            @Override
            public Integer next() {
                return fib(i++);
            }
        };
    }
}
```

Si modifichi la classe FiboSequence in modo che i numeri di Fibonacci generati sottostiano ad un meccanismo di caching che ne allevia il costo computazionale memorizzando il risultato di ogni passo di ricorsione, in modo che ogni computazione successiva con il medesimo input costi solamente un accesso in lettura alla cache. Ogni istanza della classe di FiboSequence deve possedere la propria cache. Si utilizzino liberamente le mappe del JDK.

```
//(b) 4 punti:
public static class CachedFiboSequence extends FiboSequence{
    private final Map<Integer,Integer> cache;

    CachedFiboSequence(Integer len) {
        super(len);
        this.cache = new HashMap<>();
    }

    //mi serve per la sottoclasse principalmente
    CachedFiboSequence(Integer len, Map<Integer,Integer> cache){
        super(len);
        this.cache = cache;
    }

    @Override
    protected int fib( int n ){
        if(n < 2)
            return 1;
        else{
            Integer r = cache.get(n);
            if(r == null){
                r = fib(n-1)+fib(n-2);
                cache.put(n,r);
```

```
                }
                return r;
            }
        }
    }
```

Si modifichi la classe FiboSequence in modo che la cache sia condivisa tra molteplici istanze.

```
    //(c) 2 punti:
    public static class GlobalCachedFiboSequence extends CachedFiboSequence{
        //static non può essere null, va subito inizializzata
        private final static Map<Integer,Integer> globalCache = new HashMap<>();

        GlobalCachedFiboSequence(int len){
           super(len, globalCache);
        }
    }
```

1.11.4 SkippableArrayList - esame del 1/7/2022

Si definisca una interfaccia funzionale di nome Predicate specializzando l'interfaccia generica java.util.Function del JDK in modo che il dominio sia un generic T ed il codominio sia Boolean.

```
// (a) 2 punti:

public static interface Predicate<T> extends Function<T, Boolean> {}
```

Si definisca una interfaccia Either parametrica su un tipo generico T e che definisce due metodi.

1. Il primo metodo, di nome onSuccess, prende un T e ritorna un T e viene chiamato dall'iteratore quando il predicato ha successo.
2. Il secondo metodo, di nome onFailure, viene invocato invece quando il predicato fallisce, prende un argomento di tipo T e non produce alcun risultato, tuttavia può lanciare una eccezione di tipo Exception.

```
// (B) 2 punti:

public static interface Either<T>{
 public T onSuccess(T x);
 public void onFailure(T x) throws Exception;
}
```

Si definisca la sottoclasse SkippableArrayList parametrica su un tipo E e si implementi un metodo pubblico avente firma:

```
Iterator<E> iterator(Predicate<E> p, Either<E> f)
```

che crea un iteratore con le caratteristiche accennate sopra. Più precisamente:

1. l'iteratore parte sempre dall'inizio della collezione ed arriva alla fine, andando avanti di un elemento alla volta normalmente;
2. ad ogni passo l'iteratore applica il predicato p all'elemento di tipo T corrente, che chiameremo x:
   (a) se p(x) computa true allora viene invocato il metodo onSuccess di f e passato l'elemento x come argomento;
   (b) altrimenti viene invocato il metodo onFailure e passato x come argomento a quest'ultimo;
3. l'invocazione di onFailure deve essere racchiusa dentro un blocco che assicura il trapping delle eccezioni: in altre parole, una eccezione proveniente dall'invocazione di onFailure non deve interrompere l'iteratore;
4. quando viene invocato onSuccess, il suo risultato viene restituito come elemento corrente dall'iteratore;
5. quando viene invocato onFailure, l'iteratore ritorna l'elemento originale che ha fatto fallire il predicato.

```

// (C) 6 punti:

public static class SkippableArrayList<E> extends ArrayList<E> {

 public Iterator<E> iterator(Predicate<E> p, Either<E> f){
     final Iterator<E> it = super.iterator();
     return new Iterator<E>() {

         @Override
         public boolean hasNext() {
             return it.hasNext();
         }
         @Override
         public E next() {
             E x = it.next();
             if(p.apply(x)){
                 return f.onSuccess(x);
             }else{
                 try {
                     f.onFailure(x);
```

```
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }

                return x;
            }

        }
    };
 }
}

//MAIN DEL PROF

public static void main(String[] args) {
 ArrayList<Integer> dst = new ArrayList<>();
 SkippableArrayList<Integer> src = new SkippableArrayList<>();

 Random rand = new Random();
 rand.setSeed(System.currentTimeMillis());
 for(int i = 0; i < 10; ++i){

     Integer elem = abs(rand.nextInt()%11);
     src.add(elem);
     System.out.println(elem);
 }

 System.out.println("\n");

 Predicate<Integer> p = (Integer x)->(x > 5);

 Either<Integer> e = new Either<Integer>() {
     @Override
     public Integer onSuccess(Integer x) {
         x++;
         return x;
     }

     @Override
     public void onFailure(Integer x) throws Exception {
         dst.add(x);
     }
 };

 Iterator<Integer> it = src.iterator(p,e);

 while(it.hasNext()){
     System.out.println(it.next());
 }

}
```

1.11.5 Figure geometriche - esame del 3/6/2022

Parte 1

Definiamo in Java 8+ un sistema di classi e interfacce che rappresentano figure geometriche piane e solide. Le figure geometriche rappresentate non sono posizionate nel piano cartesiano o nello spazio, sono pertanto prive di coordinate. Per semplicità esse contengono solamente le informazioni sulla lunghezza dei lati o delle facce di cui sono costituite. Prima di cominciare, realizziamo una piccola libreria interna che consiste in alcuni metodi statici generici altamente riusabili. Sia data la funzione di ordine superiore fold1 implementata tramite un metodo statico pubblico:

```
public static <T, State> State fold(Iterable<T> i, final State st0, BiFunction<State, T, State> f) {
    State st = st0;
    for (final T e : i)
        st = f.apply(st, e);
    return st;
}
```

Si implementi tramite una sola invocazione di fold() la funzione di ordine superiore sumBy avente la seguente firma:

```
    public static <T> double sumBy(Iterable<T> i, Function<T, Double> f)
```

Essa calcola la sommatoria di tutti gli elementi di i trasformandoli in double tramite f. La utilizzeremo ad esempio per calcolare il perimetro di un poligono sommando la lunghezza di tutti i suoi lati, oppure l'area laterale totale di un solido sommando l'area di tutte le superfici piane di cui è costituito.

```
// (a) 1 punti:

    public static <T> double sumBy(Iterable<T> iterable, Function<T, Double> f){
        Double st0 = 0.;

        return fold(iterable, st0, (x, y) -> x + f.apply(y));

    }
```

Avremo bisogno di ordinare le nostre figure geometriche sulla base di diversi criteri, ad esempio l'area o il volume. Si implementi il metodo statico compareBy() avente la seguente firma:

```
            public static <T> int compareBy(T s1, T s2, Function<T, Double> f);
```

Esso confronta s1 ed s2 conventerdoli prima in double tramite f, riducendo pertanto il confronto al confronto tra due numeri double.

```
// (b) 1 punti:
public static <T> int compareBy(T s1, T s2, Function<T, Double> f){
    Double double_s1 = f.apply(s1);
    Double double_s2 = f.apply(s2);
    // mia soluzione
    //return double_s1.compareTo(double_s2);
    // soluzione del prof
    return Double.compare(double_s1,double_s2);
}
```

Parte 2

Definiamo ora i tipi essenziali per rappresentare figure geometriche piane e solide. La classe Edge rappresenta grandezze 1-dimensionali come lati di poligoni, spigoli di poliedri, segmenti e circonferenze. Si dia una implementazione di default del metodo compareTo() tramite una sola invocazione della compareBy() definita sopra che esegua il confronto tra le aree.

```
//(a) 1 punti:
public static class Edge implements Comparable<Edge> {
    private final double len;

    public Edge(double len) { this.len = len; }

    public double length() { return 2; }

    @Override
    public int compareTo(Edge s) {
        return compareBy(this, s, Edge::length);
    }
}
```

L'interfaccia Surface rappresenta figure piane qualunque, si dia una implementazione di default del metodo compareTo() tramite una sola invocazione della compareBy() definita sopra che esegua il confronto tra le aree.

```
//(b) 1 punti:
public interface Surface extends Comparable<Surface> {
    double area();
    double perimiter();
    @Override
    default int compareTo(Surface s) {
        /* DA IMPLEMENTARE */
        return compareBy(this, s, Surface::area);
    }
}
```

Il sotto-tipo Polygon rappresenta poligoni: l'interfaccia specializza Surface dando una implementazione di default al metodo perimeter() e permette anche l'iterazione dei lati di cui il poligono stesso è costituito. Si dia una implementazione di default del metodo perimeter() tramite una sola invocazione della sumBy() definita sopra.

```
//(c) 1 punti:

public interface Polygon extends Surface, Iterable<Edge> {

    @Override
    default double perimiter() {
        /* DA IMPLEMENTARE */
        return sumBy(this, Edge::length);
    }

}
```

L'interfaccia Solid rappresenta solidi qualunque: si implementi il metodo compareTo() tramite una sola invocazione della compareBy() definita sopra che esegua il confronto tra i volumi.

```
//(d) 1 punti:

public interface Solid extends Comparable<Solid> {
    double outerArea(); // area laterale totale
    double volume();
    @Override
    default int compareTo(Solid s) {
        /* DA IMPLEMENTARE */
        return compareBy(this, s, Solid::volume);
    }
}
```

Polyhedron è sottotipo di Solid e rappresenta poliedri. L'interfaccia è parametrica rispetto al sottotipo di Polygon che descrive le facce di cui il poliedro è costituito. Ad esempio, le facce di un cubo sono quadrati: una classe Cube implementerebbe pertanto l'interfaccia Polyhedron avente Square come type argument, assumendo che esista Square sottotipo di Polygon.

Un Polyhedron permette l'iterazione delle facce poligonali di cui esso è costituito, fornisce inoltre una implementazione di default del metodo outerArea() che calcola semplicemente la sommatoria delle aree delle sue facce. Si implementi il metodo outerArea() tramite una sola invocazione della sumBy() definita sopra.

```
//(e) 1 punti :

public interface Polyhedron<P extends Polygon> extends Solid, Iterable<P> {
    @Override
    default double outerArea() {
        /* DA IMPLEMENTARE */
        return sumBy(this, P::area);
    }
}
```

Parte 3

Si proceda ora alla definizione di una gerarchia di classi che rappresentano figure geometriche specifiche implementando le interfacce fin qui introdotte. (a)Si implementi una classe che rappresenta sfere immutabili avente nome Sphere e che implementa l'interfaccia Solid. Il costruttore di Sphere deve prendere come parametro solamente un double: il raggio della sfera.

```
//(a) 2 punti:
public class Sphere implements Solid{

    private final double r;

    Sphere(double r){
        this.r = r;
    }

    @Override
    public double outerArea(){
        return 4 *Math.PI * Math.pow(r,2) ;
    }

    @Override
    public double volume(){
        return 4./3. * Math.PI * Math.pow(r,3);
    }
}
```

Si implementi una classe che rappresenta rettangoli immutabili avente nome Rectangle e che implementa l'interfaccia Polygon. Il costruttore di Rectangle deve prendere come parametri due double: base e altezza.

```
//(c) 2 punti:

public static class Rectangle implements Polygon{
    private final double base,altezza;

    Rectangle(double base, double altezza){
        this.base = base;
        this.altezza = altezza;
    }

    @Override
    public double area(){
        return base * altezza;
    }

    @Override
    public Iterator<Edge> iterator(){
        Edge b = new Edge(base);
        Edge h = new Edge(altezza);

        ArrayList<Edge> list = new ArrayList<>( Arrays.asList(b,h,b,h) );

        return list.iterator();
    }
}
```

Si implementi una classe che rappresenta quadrati immutabili avente nome Square e che estende Rectangle. Il costruttore di Square deve prendere un solo parametro di tipo double: il lato.

```
// (d) 2 punti:
public class Square extends Rectangle{

    Square( double lato){
        super(lato,lato);
    }
}
```

Parte 4

Si prenda in considerazione la seguente classe che rappresenta parallelepipedi immutabili. I parallelepipedi sono poliedri aventi facce rettangolari.

```
//(a) 1 punti: Si implementi il metodo volume()

public static class Parallelepiped implements Polyhedron<Rectangle> {
    protected final double width, height, depth;
    public Parallelepiped(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
    @Override
    public double volume() {
        /* DA IMPLEMENTARE */
        return width*height*depth;
    }
    @Override
    public Iterator<Rectangle> iterator() {
        Rectangle r1 = new Rectangle(width, height),
                  r2 = new Rectangle(width, depth),
                  r3 = new Rectangle(height, depth);
        return List.of(r1, r2, r3, r1, r2, r3).iterator();
    }
}
```

Si definisca la classe Cube come sottoclasse di Parallelepiped. Il costruttore di Cube deve prendere solamente un parametro di tipo double: la lunghezza dello spigolo.

```
//(b) 1 punti:

public static class Cube extends Parallelepiped{
    Cube(double side){
        super(side,side,side);
    }
}
```

Main del prof:

```

public static void main(String[] args) {
// 4.d
    {
        int facet_cnt = 1;
        // questo foreach non compila perché Cube è sottotipo di Iterable<Rectangle>,
        //non di Iterable<Square>
        // si badi che NON è possibile co-variare il tipo di ritorno del metodo iterator() di Cube
        //in modo che si specializzi in Iterator<Square>
        // perché è sound co-variare il tipo più esterno di un tipo parametrico, ma non il type argument
        // for (Square sq : new Cube(10.)) {

        for (Rectangle sq : new Cube(10.)) {  // così invece compilerebbe
            int side_cnt = 1;
            for (Edge e : sq) {
                System.out.printf("side #%d/%d = %f\n", side_cnt++, facet_cnt, e.length());
            }
            ++facet_cnt;
        }
```

```
20      }
21
22      // 5
23      {
24          Cube c1 = new Cube(1.), c2 = new Cube(2.);
25          Parallelepiped p1 = new Parallelepiped(1., 2., 3.), p2 = new Parallelepiped(2., 3., 4.);
26          List<Polyhedron<? extends Rectangle>> polys = new ArrayList<>(List.of(c1, c2, p1, p2));
27
28          // per testare rapidamente i risultati di queste sort, si usi debugger
29          Collections.sort(polys);                                                    // c1
30          for( Polyhedron<? extends Rectangle> e : polys){
31              System.out.println(e.outerArea());
32          }
33          Collections.sort(polys, (x, y) -> compareBy(x, y, Polyhedron::outerArea));          // c1
34          for( Polyhedron<? extends Rectangle> e : polys){
35              System.out.println(e.outerArea());
36          }
37          Collections.sort(polys, (x, y) -> compareBy(x, y, (p) -> p.outerArea()));           // c1
38          for( Polyhedron<? extends Rectangle> e : polys){
39              System.out.println(e.outerArea());
40          }
41      //Collections.sort(polys, (x, y) -> compareBy(x, y, (r) -> r.perimiter())); // non compila
42          Collections.sort(polys, (x, y) -> compareBy(x, y, new Function<>() {                // c1
43              @Override
44              public Double apply(Polyhedron<? extends Rectangle> r) {
45                  return r.volume();
46              }
47              }));
48              for( Polyhedron<? extends Rectangle> e : polys){
49                  System.out.println(e.outerArea());
50              }
51              for( Polyhedron<? extends Rectangle> e : polys){
52              System.out.println(e.volume());
53              }
54      // Collections.sort(polys, (x, y) -> Double.compare(sumBy(x, Square::perimiter), // non compila
55      // sumBy(y, Rectangle::perimiter)));
56          }
57      }
58  }
```

Chapter 2

Il Linguaggio C++

C++ è perfetto per il generic programming, nonostante il suo ideatore Bjarne Stroustrup lo volesse. C++ è stato creato nel 1979, ma da C++11(2010) i poi la community ha preso il sopravvento per lo sviluppo del linguaggio, inizialmente infatti Stroustrup ha dato l'ok alla community della famosa libreria <boost> e non si sono più fermati.

2.1 Il polimorfismo in C

2.1.1 Il preprocessore

Da C++20(2021) non è più necessario separare il file .hpp dal file .cpp, questo è successo perché prima di c++20 quando si scrivevano comandi come #include, #define, ecc. il compilatore non era a conoscenza di cosa facessero, infatti tutti i comandi che iniziano per '#' sono comandi del preprocessore di C, il cui funzionamento è solo quello di sostituire codice, un vero e proprio copia e incolla.

Alcuni esempi

```
#include <pizza.h>
```

In questo esempio il preprocessore prende la stringa 'pizza.h' e la cerca nei path degli include, poi toglie l'include e ci sostituisce il codice C che trova, il compilatore non è 'aware' di cosa sta includendo.

```
#ifndef CODE_PIZZA_H //se non è definito
#define CODE_PIZZA_H

void f();
void g();
void h();

#endif
```

All'interno dei 'file.h' il corpo è protetto da una guardia per evitare che il preprocessore ridefinisca le stesse cose.

```
#define MYMACRO ciao ragazzi sono io
```

Se utilizzassi questa macro da qualche parte nel mio codice il preprocessore sostituirebbe 'MYMACRO' e ci incollerebbe 'ciao ragazzi sono io'.

2.1.2 Le macro per il polimorfismo

Prendiamo per esempio di dover fare una classica funzione che prende due interi e ne scambia i valori.

```
void swap_int(int* a, int* b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Se volessi avere una funzione identica però che funzioni con i Double dovrei riscriverla, e dovrei anche cambiare il nome perché C non ha l'overloading.

```
void swap_double(double* a, double* b){
    double tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Le **macro** permettono di fare il polimorfismo anche in C, utilizzando una macro il preprocessore potrà sostituire tutte le occorrenze di un input con del codice C.

```
#define SWAP(T) \\
void swap_T##(T* a, T* b){
//il doppio # mi dice 'sostituiscimi la T anche se non è isolata'
    T tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Il preprocessore sostituisce tutte le occorrenze di T con quello che gli daremo in input, in questo modo la funzione swap funzionerà con int, double, long e qualsiasi tipo le daremo in input. Dato che il compilatore non è 'aware' potremmo anche dare in input a SWAP dei tipi problematici, che verranno poi sostituiti dal preprocessore.

```
SWAP(int)
SWAP(double)
SWAP(long)
SWAP(mio nonno)
```

In questo esempio sarà solo il compilatore a crashare quando elaborerà il tipo 'mio nonno'.

2.2 L'Object System di C++

D'ora in avanti utilizzeremo C++20, questa versione infatti ha innovato molto il linguaggio, specialmente per la funzione del preprocessore.

2.2.1 Implementazione della classe animal

In questa sezione implementeremo la classe animal in C++20 e ne commenteremo le funzionalità.

codice della classe

```
export module zoo;

import <string>;

export namespace zoo
{
    export class animal
    {
    protected:
        int weight_;
    public:
        explicit animal(int w) : weight_(w){}  //costruttore custom
        virtual ~animal(){} //distruttore



        animal(const animal& a) : weight_(a.weight_) {} //copy constructor


        virtual void eat(const animal* a){
            weight_ += a->weight_;
        }

```

```
24          const int& weight() const{ return weight_;}
25          int& weight() {return weight_;}
26
27      };
28
29      animal f(animal a){ return a;}
```

La keyword virtual

```
1   export class animal{...}
```

Il comando 'export' esiste solo in C++20 e indica che la classe è visibile dall'esterno.

```
1   export class animal
2       {
3       protected:
4           int weight_;
5       public:
6           explicit animal(int w) : weight_(w){}
7       ...
```

Il costruttore è un costruttore custom un-ario, cioè con un singolo parametro, in C++ i costruttori un-ari vengono automaticamente promossi a **conversion constructor**, per bloccare questa conversione automatica è necessario inserire la keyword **explicit**. Non c'è la keyword this perché viene invocato il copy constructor di int. Esempio:

```
1   f(5);
```

La nostra funzione f prende in input un 'animal', in presenza della keyword explicit **non compila**, altrimenti converte il costruttore di animal in un copy-constructor di int e il precedente codice compilerebbe.

```
1   virtual ~animal(){}
```

In questo distruttore la keyword **virtual** indica che la funzione è 'overridabile' dalle sottoclassi'. In c++ **non** è possibile fare l'override di tutte le funzioni, al contrario di Java, la keyword virtual è il segnale che permette ai figli di fare l'override.

2.2.2 Il copy constructor e i Binding

In C++ non si può dichiarare un oggetto se non è presente un default constructor, infatti nel default constructor può esserci scritta qualsiasi cosa, al compilatore basta che ci sia.

```
1   animal e; //DA' ERRORE
```

I binding in C++ avvengono solo quando utilizziamo le reference, tutto il resto degli assegnamenti utilizzano il copy constructor.

```
1   //COPY CONSTRUCTOR
2   animal a(4);
3   animal* b = new animal(4);
4   animal c(a);
5   animal d(f(c));
6   animal u(a);
7   //BINDING
8   animal& w = a;
```

2.3 Generic Programming in C++

C++ è un linguaggio **imperativo** con 'anche' gli oggetti. Il generic programming in c++ si avvicina agli oggetti ma non ha il subtyping. Per fare generic programming c++ utilizza i template, il sistema dei template sembra polimorfismo, ma in realtà non esiste subtyping e sono generativi, ovvero è un sistema di macro automatizzato.

```
template <class T>
T sum(const std::vector<T> &v){ //funziona SOLO con i vector, non c'è subsuction
    if(v.size() > 0)
    {
        T r(v[0]);
        for(size_t i = 0; i < v.size(); ++i)
            r += v[i]; //sottoindendo che debba esistere l'operatore +=
        return r;
    }
     //ti ritorno by value qualunque cosa il costruttore di questo tipo T ritorna come valore nullo
    return T();
}
```

Questo esempuo fa una sum cumulativa di un vector di T, dato che non c'è subsuction questo esempio funziona esclusivamente con i vector.

2.3.1 Member Types

Vogliamo che sum funzioni anche con un qualsiasi container, non solo vector. Per fare questo definiamo un template C, che rappresenta il nostro container, e ci prendiamo il value type o il reference type contenuto in C. La keyword **typename** serve per indicare al compilatore che quel pezzo di codice è un tipo.

```
typename C::value_type sum(const C& v){
    if(v.size() > 0)
    {
        typename C::value_type r(v[0]);
        for(size_t i = 0; i < v.size(); ++i){
            typename C::reference x(v[i]);
            //binding by reference senza copia temporaneo di quello che leggo in v[i]

            /*fare direttamente ::reference è più generico, potrei anche fare il reference di un
            value_type, ma se lo faccio quel reference non è effettivamente un reference
            ma sarà uno smart pointer o altra roba*/

            r += x; //sottointendo che debba esistere l'operatore +=
        }
        return r;
    }
    return typename C::value_type();
    //value_type() è un alias del template parameter C
}

int main(){
    std::vector<int> v1{1 , 2 , 3};
    int y  = sum(v1);
}
```

2.3.2 Iteratori

Le guideline di STL indicano di utilizzare gli **iteratori** per templatizzare e non i container, perché? con la sum fatta in precedenza stiamo assumendo che il container abbia l'operatore di parentesi quadre [], che non è sempre garantita, vector li ha ma per esempio i set non ce l'hanno. In Java il random access accessor [] non esiste perché ci sono i getter e setter, ma in Java posso farmi dare l'iteratore e sono sicuro che il container su cui opero ce l'abbia In c++ gli iteratori sono modellati sui pointer, cioè a immagine e somiglianza dei pointer.

Pensiamo a cosa sono i pointer: possono essere dereferenziati, supportano la somma eterogenea con un tipo(sizeof), l'assegnamento, l'incremento, il preincremento,.. Tutta questa roba c'è da 50 anni in C, C++ permette di creare degli "aggeggi" che sembrano dei pointer, ma in realtà sono iteratori, e la lettura si fa col dereference.

Per fare la sum cumulativa di un container utilizzeremo un template chiamato InputIterator, lo scriviamo così per far capire a chi legge che quello che scrivo deve essere compatibile col concept di input iterator, quindi assumo che abbia copy constructor, che il suo value_type sia costruibile col costruttore di default, che abbia l'operatore ! = omogeneo su due tipi InputIterator, l'incremento, il + = binario con a sinistra un value_type e a destra un reference (sempre di InputIterator).

Concept

Gli InputIterator sono stati definiti dalla libreria Boost(pre c++11) e sono dei **concept**: i concept esistono solo all'interno delle documentazioni e sono tutti i **requirements** che strutture molto grandi devono avere(operatori e metodi che devono funzionare perché un InputIterator funzioni), i requirements sono loro stessi dei concept(vedi **link**).

```
1  #include <vector>
2  template <class InputIterator>
3  //vogliamo rifare la funzione sum ma con gli iteratori
4  typename std::iterator_traits<InputIterator>::value_type sum(InputIterator a, InputIterator b)
5  {
6      typename std::iterator_traits<InputIterator>::value_type r;
7      //potrei fare un while(a < b)? NO, non è garantito che ci sia l'operatore < su InputIterator
8      //vedi link
9      while(a != b){
10         r += *a++;
11     }
12     return r;
13 }
14 /*
15 * Come scrivere la stessa cosa in c++14?
16 * AUTO toglie tutte le rogne di prima
17 *
18 * a patto che tu abbia scritto 'auto' prima puoi definire meglio i tipi con -> decltype(...)
19 * il senso è quello di avere il tipo di ritorno come in matematica (es. R->R),
20 * decltype nel nostro caso ci dice:
21 * 'il tipo di ritorno è quello del dereference dell'iteratore a '
22 * */
23 template <class InputIterator>
24 auto sum(InputIterator a, InputIterator b) -> decltype(*a)
25 {
26     auto r;
27     while(a != b)
28     {
29         r += *a++;
30     }
31     return r;
32
33     int k = 9;
34 }
```

Template Metaprogramming e iterator_traits

std::iterator_traits indica che se passo alla funzione qualcosa che non abbia tutti i requirements di iterator il compilatore li aggiunge, quindi tratto il Template InputIterator come un iterator e aggiungo i requirements necessari

(solo se già non li ha), questa roba si chiama **Template Metaprogramming**, che non vuol dire usare i template. es. posso passare un pointer alla funzione sum per fare la somma di un array di char.

```
int main(){
    std::vector<int> v1{1 , 2 , 3};
    int n = sum(v1.begin(),v1.end());

    char a[10];
    int m = sum(a, a+10);

}
```

2.4 Lambdas in c++

Le lambdas in c++ esistono, non essendoci Function, Runnable, Bifunction, ecc. come in c++ templatizziamo una funzione che poi verrà sostituita dal compilatore con una lambda.

```
#include <iostream>
#include <vector>
/*
 * Vogliamo fare la sum che prende in input una function
 *
 * f deve essere un function pointer con tipo di ritorno InputIterator::value_type
 * */


template <class InputIterator>
auto sum(InputIterator first, InputIterator last,
         typename InputIterator::value_type (*f)
         //f ha tipo di ritorno InputIterator::value_type
            (typename InputIterator::value_type, typename InputIterator::value_type)
            //tipi dei parametri in ingresso
        )
{
    auto r(*first);
    while(first != last)
    {
        r = f(r, *first++);
    }
    return r;
}

/* non c'è modo di passare a una funzione un function pointer --> mi serve un altro template
 * parameter
 * in questo modo posso passare anche una lambda, una funzione globale o un oggetto
 * che supporta l'overload di ()
 *
 * */

template <class InputIterator, class BigFun>
auto sum(InputIterator first, InputIterator last, BigFun f)
{
    auto r(*first);
    while(first != last)
    {
        r = f(r, *first++); //quando passerò una funzione andrà a sostituire
    }
    return r;
}

template <class T>
int global_plus(T a, T b){return a + b;}



int main(){
    std::vector<int> v1 {1, 2, 3};
    int s1 = sum(v1.begin(), v1.end(), [](int a, int b){return a + b;});
```

```
52
53          //posso scrivere anche cosi per specificare il tipo di ritorno
54          int s2 = sum(v1.begin(), v1.end(), [](int a, int b) -> int {return a + b;});
55          /*
56           * tra le parentesi [] ci va la capture/closure --> indica una lambda che si porta dietro
57           * il suo scope
58           *
59           * link : https://en.cppreference.com/w/cpp/language/lambda
60           */
61          int s3 = sum(v1.begin(),v1.end(), global_plus); //gli passo una funzione globale
62
63          std::cout<<s1<<"\n";
64          std::cout<<s2<<"\n";
65          std::cout<<s3<<"\n";
66
67      }
```

2.4.1 Alcune lambdas

```
1       {
2           // da int a int
3           auto f1 = [](int x) { return x + 1; };
4
5           // con auto sul lambda parametro
6           auto f2 = [](auto x) { return x + 1; };
7
8           // con annotazione esplicita del tipo di ritorno ed auto nel lambda parametro
9           auto f3 = [](auto x) -> int { return x + 1; };
10
11          // con annotazione esplicita sia del tipo del lambda parametro che del tipo di ritorno
12          auto f4 = [](int x) -> int { return x + 1; };
13      }
14
15      // altri esempi con reference
16      {
17          // con un const int& come lambda parametro
18          auto f1 = [](const int& x) { return x + 1; };
19
20          // auto può essere usato insieme a const e reference:
21          // il compilatore non inferisce mai & e const con auto, inferisce solo il tipo principale
22          auto f2 = [](const auto& x) { return x + 1; };
23
24          // come reference non-const
25          auto f3 = [](auto& x) { x++; };
26      }
27
28      // esempi di capture: si chiamano capture le variabili catturate dalla chiusura della lambda
29      // c++ permette di customizzare il comportamento delle capture in maniera molto fine
30      {
31          int k = 5;
32          vector<int> v{ 1, 2, 3, 4, 5 };
33
34          // v e k sono catturate per COPIA nella chiusura della lambda
35          auto f1 = [=](int x) { return x + v[0] + k; };
36
37          // v e k sono catturate per REFERENCE nella chiusura della lambda
38          auto f2 = [&](int x) { return x + v[0] + k; };
39
40          // tutto per copia (cioè solo k, nel nostro caso) eccetto v per reference
41          auto f3 = [=, &v](int x) { return x + v[0] + k; };
42
43          // tutto per reference (cioè solo v, nel nostro caso) eccetto k per copia
44          auto f4 = [&, k](int x) { return x + v[0] + k; };
45
46          // tutto per copia con rebinding dei nomi: v si chiama a e k si chiama b
47          auto f5 = [a = v, b = k](int x) { return x + a[0] + b; };
48
49          // v si chiama a ed è per reference; k si chiama b ed è per copia
```

```
        auto f6 = [&a = v, b = k](int x) { return x + a[0] + b; };
    }
```

2.5 Smart Pointer

```

/*
    * Come sappiamo c++ NON ha il Garbage Collector, quindi bisogna preoccuparsi delle delete
    *
    * La classe smart_ptr ha un counter che viene incrementato e decrementato
    * se viene creato o distrutto un oggetto
    *
    * il distruttore ~smart_ptr() decrementa il counter,
    * e solo se il counter arriva a 0 fa la delete dei suoi campi
    * */


// definiamo un piccolo type trait che ci dice se possiamo usare la delete[] per un certo tipo
template <class T, class = void>
struct can_delete
{
    static constexpr bool value = false;
};

// questa è una specializzazione parziale di can_delete che il compilatore sceglie solamente
//se compila il decltype nel secondo template argument
template <class T>
struct can_delete<T, std::void_t<decltype(delete[] declval<T*>())>>
{
    static constexpr bool value = true;
};

// uno smart points ha un tipo a cui punta ed una dimensiona statica templatizzata,
// il cui valore di default 1 se un non-array altrimenti la lunghezza del suo extent

// NOTA: si chiama extent la lunghezza statica degli array,
//ad esempio la parte tra parentesi quadre in: int[10]
export
template <class Ty, size_t L = (std::is_array_v<Ty> ? std::extent_v<Ty> : 1)>
class smart_ptr
{
private:
    using T = std::remove_extent_t<Ty>;
    using self = smart_ptr<Ty, L>;

protected:
    T* pt;
    ptrdiff_t offset;
    size_t* cnt;


    void dec()
    {
        --(*cnt);
        if (*cnt == 0)
        {
            if constexpr (can_delete<T>::value) delete[] pt;
            else delete pt;
            delete cnt;
        }
    }

    void inc()
    {
        ++(*cnt);
    }

```

```
63
64  public:
65      smart_ptr(T* pt_, ptrdiff_t offset_, size_t* cnt_)
66              : pt(pt_), offset(offset_), cnt(cnt_)
67      {
68          assert(offset >= 0 && offset < L);
69          inc();
70      }
71
72      explicit smart_ptr(T* p) : smart_ptr(p, 0, new size_t(1)) {}
73
74      smart_ptr(const self& p)
75              : pt(p.pt), cnt(p.cnt), offset(p.offset)
76      {
77          inc();
78      }
79
80      ~smart_ptr()
81      {
82          dec();
83      }
84
85      /*
86          * Quando assegno aumento il counter di quello vecchio di sinistra e
87          * quello nuovo di sinistra
88          * */
89      self& operator=(const self& p)
90      {
91          if (pt != p.pt) //controllo se gli address sono diversi --> l'address mi fa da chiave
92          {
93              dec();
94              pt = p.pt;
95              cnt = p.cnt;
96              offset = p.offset;
97              inc();
98          }
99          return *this;
100     }
101
102     T& operator*() //operatore di de-reference -> ritorna una reference del tipo T
103     {
104         //return *pt
105         return const_cast<T&>(*std::as_const(*this));
106         //*this --> dereferenzio ma in realtà ritorno per reference
107     }
108
109     const T& operator*() const //se ho uno smart pointer const --> propago la const-ness
110     {
111         //return *pt
112         return pt[offset]; //l'operatore di quadre somma e dereferenzia
113     }
114
115     bool operator==(const self& p) const
116     {
117         //return pt == p.pt
118         return pt == p.pt && offset == p.offset;
119     }
120
121     bool operator!=(const self& p) const
122     {
123         return !(*this == p);
124     }
125     /* operatore di conversione--> non casta niente, converte this con un T-pointer
126         * il tipo della conversione è il NOME dell'operatore, in questo caso T*
127         * */
128     operator T* ()
129     {
130         //return pt
131         return const_cast<T*>(std::as_const(*this).operator const T * ());
132     }
133
```

```
134     operator const T* () const
135     {
136         //return pt
137         return pt + offset;
138     }
139 
140     //using self = smart_ptr<Ty, L>; è definito sopra
141 
142     self operator+(ptrdiff_t d) const
143     {
144         //return smart_ptr<T>(pt + d);
145         return self(pt, offset + d, cnt);
146         //fa un nuovo smart pointer, non modifica uno esistente es. a = b+c
147     }
148 
149     self operator-(ptrdiff_t d) const
150     {
151         return *this + (-d);
152     }
153 
154     self& operator+=(ptrdiff_t d)
155     {
156         assert(offset + d >= 0 && offset + d < L);
157         offset += d;
158         return *this;
159     }
160 
161     self& operator-=(ptrdiff_t d)
162     {
163         return *this += -d;
164     }
165 
166     self& operator++() //++ dietro
167     {
168         return *this += 1;
169     }
170 
171     self operator++(int) //++ davanti
172     {
173         self r(*this);
174         ++(*this);
175         return r;
176     }
177 
178     self& operator--()
179     {
180         return *this -= 1;
181     }
182 
183     self operator--(int)
184     {
185         self r(*this);
186         --(*this);
187         return r;
188     }
189 
190     T* operator->()
191     {
192         return const_cast<T*>(std::as_const(*this).operator->());
193     }
194 
195     const T* operator->() const
196     {
197         return pt + offset;
198     }
199 
200     T& operator[](ptrdiff_t i)
201     {
202         return const_cast<T&>(std::as_const(*this)[i]);
203     }
204 
```

```
205     const T& operator[](ptrdiff_t i) const
206     {
207         assert(offset + i >= 0 && offset + i < L);
208         return pt[offset + i];
209     }
210 };
211
212
213
214 // usa tutti gli operatori utilizzabili per i pointer classici
215 // è templatizzata così è possibile utilizzarla con tipi qualunque,
216 // basta che implementino gli operatori richiesti
217 // la usiamo per testare gli smart pointer
218 template <class Pointer>
219 void demo(Pointer p)
220 {
221     Pointer p2(p);
222     p = p2;
223     *p;          // in C, C++ ed altri linguaggi imperativi non è necessario utilizzare
224             //il risultato di una chiamata a funzione o operatore
225     const Pointer p3(p + 2);
226     p2 = p + 2;
227     ++p2;
228     p++;
229     --p2;
230     p += 1;
231     p -= 2;
232     p3[0];
233 }
234
235
236 export void test()
237 {
238     int* a = new int[10];
239     demo(a);
240
241     smart_ptr<int[5]> a2(new int[5]);
242     demo(a2);
243
244     smart_ptr<int, 10> b(a);
245     demo(b);
246
247     smart_ptr<double[10]> d(new double[10]);
248     demo(d);
249
250 }
```

2.6 Esercizi

2.6.1 Alberi

```cpp
#include <vector>

using namespace std;

template<class T>
class TreeNode {
private:
    T data;
    TreeNode *left;
    TreeNode *right;
    vector<T> dfs = new vector<T>;

    void pushDfs(vector<T> &x) {
        x.push_back(this->data);
        if (this->left != nullptr) {
            this->left->pushDfs(x);
        }
        if (this->right != nullptr) {
            this->right->pushDfs(x);
        }
    }

public:

    using iterator = typename vector<T>::iterator; // typedef typename vector<T>::iterator iterator;
    using const_iterator = typename vector<T>::const_iterator;

    TreeNode(T data_, TreeNode *left_, TreeNode *right_) : data(data_), left(left_), right(right_) {
        pushDfs(dfs);
    };


    TreeNode& operator=(const TreeNode &old) {
        if(this==old){
            return *this;
        }
        this->dfs = old.dfs;
        this->data=old.data;
        this->left=old.left;
        this->right=old.right;
        return *this;
    }
    ~TreeNode(){
        if(left!= nullptr){
            delete left;
        }
        if(right!= nullptr){
            delete right;
        }
        delete data;
        delete dfs;
    }

    bool operator!=(const TreeNode &test) const {
        return !(this==test);
    }

    bool operator==(const TreeNode &test) const {
        if (!this && !test) {
            return true;
        }
        if (this != nullptr && test == nullptr) {
            return false;
        }
```

```
65          if (this == nullptr && test != nullptr) {
66              return false;
67          }
68          if (data != test.data) {
69              return false;
70          }
71          return left == test.left && right == test.right;
72      }
73
74      iterator begin() {
75          return dfs.begin();
76      }
77
78      iterator end() {
79          return dfs.end();
80      }
81
82      const_iterator begin() const {
83          return dfs.cbegin();
84      }
85
86      const_iterator end() const {
87          return dfs.cend();
88      }
89      static TreeNode<T> foglia(const T& data){
90          return new TreeNode<T>(data, nullptr, nullptr);
91      }
92
93  };
94
95
96  template <class T>
97  TreeNode<T> * sx(T data, const TreeNode<T>& sx){
98      return new TreeNode(data,sx, nullptr);
99  }
```

2.6.2 Matrici

```
#include<vector>
#include<iostream>

template<class T>
class matrix {
private:
    std::vector<T> v;
    size_t rows;
    size_t cols;
public:
    matrix() : v(), rows(0), cols(0) {};

    matrix(size_t n, size_t m) : v(n * m), rows(n), cols(m) {};

    matrix(size_t n, size_t m, T num) : v(n * m, num), rows(n), cols(m) {};

    matrix(const matrix &source) : v(source.v), rows(source.rows), cols(source.cols) {};

    ~matrix() {};

    using iterator = typename std::vector<T>::iterator;
    using const_iterator = typename std::vector<T>::const_iterator;
    using value_type = T;

    T &operator()(const size_t &n, const size_t &m) {
        return v.at(rows * n + m);
    }

    T &operator()(const size_t &n, const size_t &m) const {
        return v.at(rows * n + m);
    }

    matrix &operator=(const matrix &source) {
        if (source != *this) {
            this->v(source.v);
            this->rows(source.rows);
            this->cols(source.cols);
        }

        return *this;
    }

    iterator begin() {
        return v.begin();
    }

    const_iterator begin() const {
        return v.cbegin();
    }

    iterator end() {
        return v.end();
    }

    const_iterator end() const {
        return v.cend();
    }

    template<class fun>
    T &sum(fun f) {
        iterator begin = this->begin();
        iterator end = this->end();
        T ret(*(begin++));

        while (begin != end) {
            ret = f(ret, *(begin++));
        }
        return ret;
```

```
69      }
70  };
71
72  //operatore globale, prende in input una matrice
73  template<class T>
74  std::ostream &operator<<(std::ostream &os, const matrix<T> &m) {
75      auto begin = m.begin();
76      auto end = m.end();
77
78      while (begin != end) {
79          os << *(begin++);
80      }
81      return os;
82  }
83
84
85  int main() {
86      matrix<int> m(5, 5, 5);
87
88      int sum = m.sum([](int a, int b) { return a + b; });
89
90      std::cout << sum << "\n";
91      std::cout << m;
92  }
```

2.6.3 Matrici del prof

```cpp
#include <iostream>
#include <vector>

using namespace std;

// 6
template <class T>
class matrix
{
private:
    size_t cols;
    vector<T> v;

public:
    matrix() : cols(0), v() {}
    matrix(size_t rows, size_t cols_) : cols(cols_), v(rows * cols) {}
    matrix(size_t rows, size_t cols_, const T& v) : cols(cols_), v(rows* cols, v) {}
    matrix(const matrix<T>& m) : cols(m.cols), v(m.v) {}

    typedef T value_type;
    typedef typename vector<T>::iterator iterator;
    typedef typename vector<T>::const_iterator const_iterator;

    matrix<T>& operator=(const matrix<T>& m)
    {
        v = m.v;
        return *this;
    }

    T& operator()(size_t i, size_t j)
    {
        return v[i * cols + j];
    }

    const T& operator()(size_t i, size_t j) const
    {
        return (*this)(i, j);
    }

    iterator begin()
    {
        return v.begin();
    }

    iterator end()
    {
        return v.end();
    }

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

    const_iterator end() const
    {
        return end();
    }
};


int main()
{
    matrix<double> m1;          // non inizializzata
    matrix<double> m2(10, 20);  // 10*20 inizializzata col default constructor di double
    matrix<double> m3(m2);      // costruita per copia
    m1 = m2;                    // assegnamento
    m3(3, 1) = 11.23;           // operatore di accesso come left-value
```

```
69
70      for (typename matrix<double>::iterator it = m1.begin(); it != m1.end(); ++it) {
71          typename matrix<double>::value_type& x = *it;   // de-reference non-const
72          x = m2(0, 2);                                                          // operatore di acces
73      }
74
75      matrix<string> ms(5, 4, "ciao"); // 5*4 inizializzata col la stringa passata come terzo argomento
76      for (typename matrix<string>::const_iterator it = ms.begin(); it != ms.end(); ++it)
77          cout << *it;                            // de-reference const
78  }
```

2.6.4 Curve

```
#include <functional>
#include <iostream>
#include <utility>

using namespace std;

using real = double;
using unary_fun = function<real(const real&)>;

#define RESOLUTION (1000)

class curve
{
private:
    real a, b;
    unary_fun f;
// risoluzione dell'intervallo [a, b]
public:
    curve(const real& a_, const real& b_, const unary_fun& f_) : f(f_), a(a_), b(b_) {}
    curve(const real& c);
    real get_dx() const { return (b - a) / RESOLUTION; }
    pair<real, real> interval() const { return pair<real, real>(a, b); }
    real operator()(const real& x) const { return f(x); }

    curve derivative() const {
        return curve( a , b , [&, dx = get_dx()] (const real &x){ return (f(x + dx) - f(x) ) / dx; });
    }

    curve primitive() const {
        return curve( a, b , [=,dx = get_dx()] (const real& x){return f(x) * dx;});
    }

    real integral() const {
        const unary_fun& F = primitive().f;

        return F(b) - F(a);
    }

    class iterator
    {
    private:
        const curve& c;
        real x;
    public:
        iterator(const curve& c_, const real& x_) : c(c_), x(x_) {}
        iterator(const iterator& c) = default;

        pair<real, real> operator*() const {
            return pair<real,real>(x,c.f(x));
        }

        iterator operator++() {
            x = x + c.get_dx();
            return *this;
        }

        iterator operator++(int) {
            iterator tmp(*this);
            ++(*this);
            return tmp;
        }


        bool operator!=(const iterator& it) const {
            //mia soluzione
            //return *(*this) != (*it);
            //soluzione del prof
            return abs(x - it.x) >= c.get_dx();
```

```
69          }
70
71      };
72      iterator begin() const {
73          return iterator(*this, a);
74      }
75      iterator end() const {
76          return iterator(*this, b + get_dx() );
77      }
78  };
79
80
81  int main(){
82      curve c(-10., 10., [](const real& x) { return x * x - 2 * x + 1; });
83      for (curve::iterator it = c.begin(); it != c.end(); ++it)
84      {
85          const pair<real, real>& p = *it;
86          const real& x = p.first, & y = p.second;
87          cout << "c(" << x << ") = " << y << endl;
88      }
89  }
```

2.6.5 Pair

```
#include <iostream>


template <class A, typename B>
class pair{

// necessario per il copy constructor templatizzato
template <class C, typename D> friend class pair;

protected:
    A first;
    B second;
public:
    pair(const A& first_, const B& second_) : first(first_), second(second_){}
    pair(const pair<A,B>& source) : first(source.first), second(source.second){}

    template <class C, typename D>
    pair(const pair<C, D>& p) : first(p.first), second(p.second) {}

    pair<A,B>& operator=( const pair<A,B>& p){
        first = p.first;
        second = p.second;
        return *this;
    }

    //pre-incremento
    pair<A,B> operator++(){
        ++first;
        ++second;
        return *this;
    }

    //post-incremento
    pair<A,B> operator++(int){
        pair<A,B> tmp(*this);
        ++(*this);
        return tmp;
    }

    //confronto
    bool operator==( const pair<A,B>& p ){
        return first == p.first && second == p.second;
    }

    bool operator!=( const pair<A,B>& p ){
        return !(*this == p);
    }

    //operatori aritmetici(gli altri sono analoghi)
    pair<A,B> operator+( const pair<A,B> p){
        return pair<A,B>(first + p.first, second + p.second);
    }

    pair<A,B>& operator+=( const pair<A,B> p){
        first += p.first;
        second += p.second;
        return *this;
    }

    //setter (const e non-const)
    const A& fst() const{
        return first;
    }

    A& fst(){
        return first;
    }

```

```
69      const B& snd() const{
70          return second;
71      }
72
73      B& snd(){
74          return second;
75      }
76
77  };
78
79  //operatore di output globale
80  template<class A, typename B>
81  std::ostream& operator<<(std::ostream &os, const pair<A,B>& p){
82      os << "first: "  << p.fst();
83      os << " second: " << p.snd();
84      return os;
85  }
86
87  int main() {
88      pair<int, int> p1(4, 5);
89      pair<int, int> p2(p1);
90
91      pair<std::string, bool> p3("ciao", true);
92      pair<double, double> p4(p1);
93
94      p1 = p2;
95
96      int n = p1.fst();
97      p1.snd() = p1.snd() * 3;
98      p4 += p1;    // converte implicitamente il RV in un pair<double, double>
99                   // tramite un conversion copy-constructor templatizzato
100
101     std::cout<<p4;
102
103     return 0;
104 }
```