0. Introduzione agli appunti

Questi appunti non forniscono in alcun modo un metodo di studio.

Il codice scritto negli esempi non é stato mai compilato, quindi potrebbe contenere errori di sintassi, la maggior parte é stata scritta durante le lezioni e mai revisionata.

Per il codice completo rimando alla repo del professore, non so sotto quale licenza sia stato reso pubblico quindi ho deciso di non copiare/correggere questi appunti con il codice da lui scritto, in ogni caso nella maggior parte dei casi il é praticamente lo stesso, se non a livello di scrittura almeno logicamente.

Il prof Spanó (che tiene il corso) **NON** é in nessun modo affiliato alla repo da cui viene questo file e di conseguenza **NON** é responsabile di eventuali errori presenti in esso (molto probabilmente non é nemmeno a conoscenza di questi appunti)

DISCLAIMER:
Mi scuso in anticipo se qualcosa dovesse risultare incompleto, il fatto é che ho iniziato a lavorare, alcune lezioni coincidevano con orari di lavoro e studiare dalle registrazioni é divertente come fare la roulette russa a caricatore pieno... (iniziamo bene)

0.1 Licenza (CC BY-SA 4.0)

1. Intro

Impareremo ad usare gli oggetti per davvero (blast #1)

sia `java` che `c++`

java é class centrico (ma questo lo sapevamo)
C++ é una bestia un po' strana e si puó usare anche senza oggetti

L'overriding in java é fondamentale

1.1 Zoo

```
public class Zoo{
    public static class Animal{
        protected int weight;

        public Animal(int weight){
            this.weight = weight;
        }

        public void eat(Animl a){
            this.weight += a.weight;
        }
    }

    public static class Dog extends Animal {
        private String hairColor;

        public Dog(int w, String hairColor){
            super(w);
            this.hairColor = haircolor;
        }

        @override
        public void eat(Animal a){
            this.weight = a.weight *2;
        }
    }

    public static void main(String[] args){
```

```
        Animal fido = new Dog(50,"Red");
        Dog pluto = new Dog(20,"Dotted");
        Animal ciccio = pluto;
        Dog bobby = pluto;
        //Dog poppy = fido; //non valido (supertipo in tipo)
        ciccio.eat(pluto); //dynamic dispatching
        /* funziona con tabella di virtualizzazione */

        ciccio = bobby;

    }
}
```

1.2 Come funziona il dynamic dispatching:

praticamente esiste una tabella per ogni oggetto che punta alle funzioni varie dell'oggetto stesso, per Animal avremo:

```
...
|eat|0x567647...|
...
```

nel dynamic dispatching andiamo a cambiare il puntatore (a runtime) in questa tabella ficcando (per esempio) quello di Dog e quindi anche se avremo subsumption chiameremo sempre le funzioni giuste!

1.3 Definizioni varie

regola numero 0 in un metodo non statico hai sempre this oltre che avere i parametri.

espressione: pezzo di codice che calcola qualcosa (x+8)
statement: pezzo di codice che fa qualcosa (y = x+8)

binding: nuova var con un valore.
assegnamento: riassegno una var vecchia

subsumption: rendere piú generale un'espressione interpretando questa tramite una classe madre (in java si puó salire fino alla classe madre di tutte Object)

Esempio:

```
Animal fido = new Dog(50,"Red");
/*
Sto facendo subsumption perché ho preso un Dog e l'ho messo in un Animal
```

```
(che é la classe madre di Dog)
*/
```

2. Generics

in questo esempio vedremo come implementare un sistema di Generics prima che Java implementasse i Generics

```
package it.unive.dais.po2.tinyJDK;

public class NoGenerics {
    public interface Iterator{
        boolean hasNext();
        Object next(); //praticamente si fa subsumption a Object
    }

    //tutte le entitá scorribili
    public interface Iterable {
        Iterator iterator();
    }

    //aggiunge un modo per mettere, togliere e vedere se ci sta
    public interface Collection extends Iterable{
        int size();
        void add(Object x);
        void remove(Object x);
        boolean contains(Object x);
    }

    //permette di fare cose con indici
    public interface List extends Collection {
        Object get(int index);
        void set(int index, Object x);
    }

    public static class ArrayList implements List{

        private Object[] a;
        private int sz=0;

        public ArrayList(int capacity){
            a = new Object[capacity];
        }
```

```
public ArrayList(){
    this(10);
}

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

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

@Override
public void remove(Object x){

}

@Override
public boolean contains(Object x) {
    return false;
}

@Override
public Iterator iterator(){
    return null;
}

@Override
public Object get(int index){
    return a[index];
}

@Override
```

```
        public void set(int index, Object x){
            a[index] = x;
        }
    }
}
```

Quello che facciamo é di fatti subsumere ad object rischiando ovviamente di spaccare tutto siccome poi sta al programmatore non sbagliare i vari cast "di ritorno"

2.1 Implementazione dei generics

Con i generics veri:

TinyJDK - Package

```
package it.unive.dais.po2.tinyJDK;

import Zoo;

public class Generics {
    // parleremo meglio degli iteratori in parte 4
    public interface Iterator<T>{
        boolean hasNext();
        T next();
    }

    //tutte le entitá scorribili
    public interface Iterable<T> {
        Iterator<T> iterator();
    }

    //aggiunge un modo per mettere, togliere e vedere se ci sta
    public interface Collection<T> extends Iterable{
        int size();
        void add(T x);
        void remove(T x);
        boolean contains(T x);
    }

    //permette di fare cose con indici
    public interface List<T> extends Collection<T> {
        T get(int index);
        void set(int index, T x);
    }
```

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

        private T[] a;
        private int sz=0;

        public ArrayList(int capacity){
            a = new T[capacity];
        }

        public ArrayList(){
            this(10);
        }

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

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

        @Override
        public void remove(Object x){
                    //potrei usare l'hashcode per trovarlo ma sarebbe
spoiler
        }

                @Override void removeAt(int index){
                    if(index < sz && index > -1){
                        a[index] = null;
                    }
                }
```

```
        @Override
        public boolean contains(Object x) {
            return false;
                            //potrei usare l'hashcode per trovarlo ma
sarebbe spoiler
        }

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

        @Override
        public T get(int index){
            return a[index];
        }

        @Override
        public void set(int index, T x){
            a[index] = x;
        }
    }

    public static void main(String[] args){
        ArrayList<Zoo.Animal> c = new ArrayList<Zoo.Animal>();

        c.add(new Zoo.Dog(3,"nero")); //posso addare un dog perché sottotipo
di Animal

        //MA
        //ArrayList<Zoo.Animal> d = new ArrayList<Zoo.Dog>()
        //NON SI PUÓ FARE PERCHÉ SE IO POI PASSO UN ANIMALE CHE NON É DOG
ROMPO TUTTO
        //
        // POSSO SUBSUMERE MA LA PARTE DENTRO LE PARENTESI ACUTE NON SI
TOCCA
    }
}
```

2.2 TIPI PARAMETRICI (sempre Generics)

f(x) -> x parametro (nome che do alla cosa che non conosco)

f(9) -> argomento (nome che do all'entitá che passo) é un'espressione

quindi la stessa roba vale per `<T>`

2.2.1 Forwarding

```
class ArrayList<T> implements List<T> {}
                |- qua é param     |- qua é arg
```

2.2.2 Type Erasure (perché non posso istanziare un generic)

raw type => tipo generic viene convertito in tipo vero a "compile time", cosí la JVM vecchia non da problemi e le banche non scassano...

praticamente quando java compila toglie qualsiasi tipo di generic e non sarebbe in grado di distinguere fra due implementazioni dello stesso generic

Esempio:

NON POSSO FARE;

```
public static class MyClass implements Iterable<String>, Iterable<Integer>{
    public void foo(Iterable<String>){
        ...
    }

    public void foo(Iterable<Integer>){
        ...
    }
}
```

togliendo i tipo generici a compile time non sa fare quell'overloading siccome per lui sono due implementazioni della stessa cosa.

come fixo?

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

```
            }
            a[sz++] = x;
        }

@Override
        public T get(int index){
            return (T) a[index];
        }
```

Faccio una cosa brutta, il mio a diventa Object e quando ritorno ri-casto a T tanto sono sicuro che la roba che ho messo dentro (siccome faccio con la add) é per forza roba T

questo perché semplicemente potrei rompere tutto se provassi a instanziare un generic che poi viene levato a runtime

quando un membro di una classe é statico non vede i type parameter della classe, per capirci:

```
public class Test <T>{
    public static T foo(){
        //questo non si puó fare
                // non posso ritornare tipo T
    }
}
```

parleremo ancora di di generics e bestialitá simili al capitolo 14. WildCards

3. Classi nested

Stanno dentro altre classi

NON SONO FIGLIE ma la classe che ospita una nested class é detta *enclosing*.

Se sono static non possono vedere i fratelli

Un esempio completo di classe con una nested:

```
//enclosing
public class ArrayList<T> implements List<T>{

        private Object[] a;
        private int sz=0;

        public ArrayList(int capacity){
            a = new T[capacity];
        }

        public ArrayList(){
            this(10);
        }

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

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

        @Override
```

```
        public void remove(Object x){
                        ...
        }

        @Override
        public boolean contains(Object x) {
            return false;
        }

                 // NESTED
        private class MyIterator implements Iterator<T>{

            private int pos = 0;

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

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

        @Override
        public Iterator<T> iterator() {
            return new MyIterator<T>();
        }

        @Override
        public T get(int index){
            return (T) a[index];
        }

        @Override
        public void set(int index, T x){
            a[index] = x;
        }
    }
```

Nota:

una classe nested puó accedere al "this" delle sue classe enclosing ma é un automatismo, volendolo

fare esplicitamente dovremmo usare

```
<nomeclasse>.this.<altro...> //de-zuccherato
```

quindi nel caso sopra sarebbe:

```
ArrayList.this.size(); //per accedere a size di ArrayList dalla nested
```

Esempio generico:

```
public class A {
    public class B1{
        //vedo B1.this e A.this
        public class C{
            //vedo C.this, B1.this e A.this
        }
    }
    public class B2{
        //vedo B2.this e A.this
    }
}
```

ovviamente per instanziarle NON posso fare:

```
A.B2 foo = new A.B2();
```

l'unico modo sarebbe rendere B2 una classe statica (ma non posso piú accedere al this di A)

4. Iteratori

Un coso con due bottoni "has next" e "next" semplicemente astrae un qualcosa di scorribile a queste due interazioni (come lo scarico del cesso, lo usi ma non sai come funzia/cosa fa realmente)

tornando seri:

posso scorrere qualsiasi roba che mi da un iteratore con un while:

```
List<Integer> l = new ArrayList();

Iterator<Integer> it = l.iteratror;

while(it.hasNext()){
    Integer i = it.next();
    //faccio roba con i
}
```

4.1 Iteratori non statici

Non statico, ovvero con classe nested dentro la classe iterabile che puó quindi accedere ai metodi e campi della classe che la ospita, detta classe enclosing.

Implementazione dentro una classe arrayList

```
//classe enclosing
public class ArrayList<T> implements List<T> {
    ...

        private class MyIterator implements Iterator<T> {
            private int pos = 0;

            //ricordo che sto robo puó accedere al this di arraylist

            @Override
            public boolean hasNext(){
                return pos < size(); //size é una funzione della classe
enclosing
            }
```

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

4.2 Iteratori statici

```
private static class myStaticIterator<E> implements Iterator<E> {

        private final ArrayList<E> a;
        private int pos = 0;

        public myStaticIterator(ArrayList<E> a) {
            this.a = a;
        }

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

        @Override
        public E next() {
            return a.get(pos++);
        }
    }
```

4.3 iteratori istanziati come anonymous class

vedi capitolo 5 (primo esempio)

di base si implementa come l'iteratore non statico e ha le stesse possibilitá.

5. Anonymous Class

Una anonymous class é una estensione che permette di istanziare al volo un'interfaccia implementando subito tutto il necessario senza dichiarazioni e nomi ulteriori.

```
@Override
public Iterator<T> iterator(){
    int sz = size();

    return new Iterator<T>(){
        private int pos = 0

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

        @Override
        public T next(){
            return get(pos++);
        }
    }; //dalla new a questo ; é una solo espressione se comprendiamo anche return si chiama statement
}
```

Il tipo in questo caso é Iterator di T.

(in c++ si chiamano anonymous objects)

5.1 Closure

Un'entitá che porta con se un pezzo di scope.

hanno un enorme vantaggio, come nel caso di `sz = size()` nell'esempio sopra tutto ció che sta peró in quel pezzo di scope deve essere final.

Effectively final

Nel caso di sz non é scritto ma di fatti la variabile "effectively" final

Infatti dalla versione 13 se un campo viene dichiarato e assegnato solo una volta tecnicamente viene considerato final.

6. Set

Un set definisce un insieme (nel senso matematico del termine), quindi:

- Non ammette duplicati
- Non definisce un ordinamento per i suoi elementi.

(Esiste anche java.util.SortedSet che mette a disposizione una versione "ordinabile" di questi)

Un implementazione di Set in ogni caso non consente l'aggiunta di un elemento all'insieme se questo è già presente al suo interno.

Il confronto tra oggetti avviene attraverso i metodi `equals()` e `hashCode()` che ogni classe eredita dalla superclasse `Object`.

Di seguito la nostra implementazione nel nostro TinyJDK:

```
public interface Set<T> extends Collection<T>{
    public int size();
    public void add(T x);
    public void remove(T x);
    public bool contains(T x);
    //tutta 'sta roba sopra la abbiamo giá da collection
    public Iterator<T> iterator();
}
```

Un'implementazione dell'interfaccia appena scritta è la seguente copia da noi scritta per la classe `HashSet`:

```
public class HashSet<T> implements Set<T> {

    private List<T> l = new ArrayList<>();

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

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

```
    @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(o);
            }
        }
    }

    @Override
    public bool contains(T x){

        for(int i=0;i<l.size();++i){
            T o = l.get(i);
            if(o.hashCode() == x.hashCode()){
                return true;
            }
        }

        return true;
    }

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

La classe `HashSet` usa come metodo di confronto fra gli elementi da inserire in un suo Set il metodo `hashCode`.

Per maggiori informazioni sul metodo hashCode puoi consultare i miei appunti su https://github.com/WAPEETY/appunti-PO1 alla sezione # 11.2 HashCode.

7. Queue

Le code, sono strutture dati con tre metodi base:

- `void push(T el)` permette di inserire un'elemento
- `T pop()` permette di prendere un elemento (lo eliminerá dalla coda)
- `T peek()` permette di prendere un elemento senza eliminerlo dalla coda

Interfaccia per il nostro TinyJDK:

```
public interface Queue<T>{
    /*NOTA: qui il prof ha esteso collection
    non ho condiviso la sua scelta in quanto porterebbe o
    ad un'estensione rotta della stessa (facendo override
    di metodi come la remove(Tx) solo per lanciare
    eccezioni e non implementarle) o ad implementare queste
    funzioni invalidando il concetto logico di Queue
    */
    void push(T x);
    T pop();
    T peek();
}
```

7.1 Quale elemento prendono `pop()` e `peak()`?

La pop e la peak come si nota dalla firma non hanno alcun parametro in ingresso, infatti non é possibile scegliere esplicitamente l'elemento che vogliamo prendere.

A seconda dell'implementazione infatti sará la coda stessa a darci un elemento, per esempio se creiamo una coda LIFO (Last In First Out) allora la `pop()` e la `peak()` vedranno solo la "fine" della pila.

Esempio di implementazione di una LIFO queue per il TinyJDK:

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

    void push(T x){
        l.add(x);
    }
    T pop(){
```

```
        T obj = this.peek();
        l.removeAt(this.size()-1);
        return obj;
    }
    T peek(){
        return l.at(this.size()-1);
    }

    int size(){
        return l.size();
    }
}
```

8. Map

Una Map é un tipo astratto che definisce una struttura dati in grado di memorizzare elementi nella forma di coppie chiave-valore.
Ogni elemento all'interno di una mappa è identificato da una determinata chiave, questa consente successivamente di recuperare l'elemento precedentemente inserito.

Da qui in poi useremo le Exception, se non sai come si usano/cosa sono vai a vedere i miei appunti alla sezione # 14. Eccezioni

Interfaccia per il TinyJDK:

```
public interface Map<K,V> extends Iterable<Pair<K,V>> {
    class KeyNotFoundException extends Exception{
        public KeyNotFoundException (String msg){
            super(msg);
        }
    }

    void put(K key, V val);
    V get(K key) throws KeyNotFoundException;
    V removeKey(K key) throws KeyNotFoundException;
}
```

```
public class HashMap<K, V> implements Map<K, V> {
    private static int CAPACITY = 1000;

    private List<List<Pair<K, V>>> l = new ArrayList<>(CAPACITY);

    @Override
    public Iterator<Pair<K, V>> iterator() {
        return new Iterator<Pair<K, V>>() {
            private int i = 0, j = -1;
            private boolean _hasNext = true;

            @Override
            public boolean hasNext() {
                return _hasNext;
            }
```

```
        @Override
        public Pair<K, V> next() {
            Pair<K, V> r = null;
            if (j >= 0) {
                List<Pair<K, V>> inl = l.get(i);
                r = inl.get(j++);
                if (j > inl.size())
                    j = -1;
            }
            else {
                for (; i < l.size(); ++i) {
                    List<Pair<K, V>> inl = l.get(i);
                    if (inl == null)
                        continue;
                    r = inl.get(0);
                    j = 1;
                }
            }
            if (r == null)
                _hasNext = false;
            return r;
        }
    };
}

@Override
public void put(K key, V value) {
    int h = key.hashCode() % CAPACITY;
    List<Pair<K, V>> inl = l.get(h);
    if (inl == null) {
        inl = new ArrayList<>();
        l.set(h, inl);
    }
    inl.add(new Pair<>(key, value));
}

@Override
public V get(K key) throws KeyNotFoundException {
    int h = key.hashCode() % CAPACITY;
    List<Pair<K, V>> inl = l.get(h);
    if (inl != null) {
        for (int i = 0; i < inl.size(); ++i) {
            Pair<K, V> p = inl.get(i);
```

```
                if (p.fst.equals(key))
                    return p.snd;
            }
        }
        throw new KeyNotFoundException(String.format("key %s is missing",
key));
//        throw new RuntimeException(String.format("unexpected error: key %s
is not found at index %d", key, h));
    }

    @Override
    public boolean containsKey(K key) {
        // TODO per casa
        return false;
    }

    @Override
    public V removeKey(K key) throws KeyNotFoundException {
        // Not  implemented
        return null;
    }
}
```

9. Binary Tree (& Fake constructors)

Gli alberi binari sono strutture abbastanza complesse a prima vista ma non sono altro che una struttura dati fortemente "ricorsiva"
offrono diversi spunti di riflessione soprattutto in questo campo, possiamo considerare come "caso base" o fine di un ramo una foglia mentre tutti gli altri casi sono semplicemente degli alberi piú piccoli.

Assodato questo quindi ogni "nodo" puó avere:

- un genitore (riferimento all'albero da cui viene)
- due figli (riferimenti algli alberi sinistro e destro che "regge")
- un dato (di un tipo parametrico T dai noi scelto).

Per comoditá nella seguente interfaccia abbiamo messo anche funzioni come la `height()` (numero di nodi da attraversare per arrivare alla radice) e la `root()` che restituisce il riferimento alla radice stessa

Interfaccia per il TinyJDK:

```
public interface BinaryTree<T> Iterable<T>{
    BinaryTree<T> left();
    BinaryTree<T> right();
    T get();
    int height();
    BinaryTree<T> parent();
    BinaryTree<T> root();
}
```

Implementazione non completa!

```
public class NodeBinaryTree<T> implements BinaryTree<T> {

    private final T data;
    private final BinaryTree<T> l, r, p;

    public NodeBinaryTree(T data, BinaryTree<T> l, BinaryTree<T> r) {
        this.data = data;
        this.l = l;
        this.r = r;
                this.p = null;
                this.root = this;
    }
```

```
        public NodeBinaryTree(T data, BinaryTree<T> l, BinaryTree<T> r,
BinaryTree<T> p) {
        this(data,l,r);
                this.p = p;
    }

        public NodeBinaryTree(T data, BinaryTree<T> l, BinaryTree<T> r,
BinaryTree<T> p,BinaryTree<T> root) {
        this(data,l,r,p);
                this.root = root;
    }

    public NodeBinaryTree(T data) {
        this(data, null, null);
    }

        @Override
    public BinaryTree<T> left() {
        return this.l;
    }

    @Override
    public BinaryTree<T> right() {
        return this.r;
    }

    @Override
    public T get() {
        return this.data;
    }

    @Override
    public int height() {
        return 0;
    }

    @Override
    public BinaryTree<T> parent() {
        return p;
    }

    @Override
```

```
    public BinaryTree<T> root() {
        return null;
    }

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

    public <T> BinaryTree<T> Parent(BinaryTree<T> p) {
            this.p = p;
    }

        public <T> BinaryTree<T> Root(BinaryTree<T> r) {
        this.root = root;
            //questo dovrebbe propagare ricorsivamente verso tutte le foglie
il nuovo root, altrimenti avremmo un'implementazione rotta!
        }
```

Problema!
Come faccio a costruire un albero passando solo la foglia a sinistra o destra? o ancora... il root o il parent? é impossibile, i costruttori non potrebbero essere messi in overload! quindi si creano i fake constructors.

I fake constructor sono funzioni statiche che fanno da wrapper per i reali costruttori e a differenza di questi hanno un nome quindi possono essere esplicitamente chiamati.

```
    public static <T> BinaryTree<T> L(T data, BinaryTree<T> l) {
        return new NodeBinaryTree<>(data, l, null);
    }

    public static <T> BinaryTree<T> R(T data, BinaryTree<T> r) {
        return new NodeBinaryTree<>(data, null, r);
    }
}
```

10. Lambda

Roba vecchia:
all'interface list abbiamo aggiunto removeif(Predicate p) che scorre tutta la lista e applica l'if (predicato) che ritorna `true` o `false`.

sarebbe bello scrivere c.removeIf(x -> x > 80)
(SPOILER: sono le lambda)

- sono parametrizzabili su x -> *espressione* oppure x-> { *statement* }

Esempio:

- Le lambda vengono trasformate dal parser in anonymous class (quindi ancora una volta java non ha implementato realmente un ca**o)

Regola:
Finché ci sta **solo un metodo** e ha **massimo un input** e **massimo un output** tutto può essere trasformato in una lambda

- ha Inferenza dei tipi, capisce il tipo dal codice.
- Le lambda possono essere scritte come
  - statement `x-> { return x>80; }`
  - come espressioni `x -> x>80`
- Tutte le var usate nelle lambda devono essere final o effective final

---

Momento meme:
La programmazione funziona in due direzioni:
Complessitá dell'Architettura
Complessità algoritmica

Quindi puoi avere roba con:

- *Architettura complessa e algoritmi semplici (software bancari)*
- *Architettura semplice e algoritmi complessi (manipolazione audio/video)*
- *Architettura complessa e algoritmi complessi (videogiochi)*
- *Architettura semplice e programmi semplici (I programmi che scrivete voi)*
  ~ cit. Spanò

10.1 Le 4 forme di lambda

10.1.1 Function (Riceve una cosa e ne ritorna un'altra)

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

10.1.2 Consumer (Riceve una cosa e non ritorna nulla)

Ma se volessi fare una lambda che non ritorna nulla?
Esempio

```
Function <Integer> g = x -> { System.out.println(x); }
```

Dovró fare una *consumer*, un tipo speciale di lambda definito cosí

```
intrerface Consumer<T>{
    void accept(T x);
}
```

10.1.3 Supplier (Riceve nulla e ritorna una cosa)

Mentre se volessi fare una che non prende nulla ma ritorna qualcosa faccio una Supplier

```
interface Supplier<T> {
    T get();
}
```

10.1.4 Runnable (Riceve nulla e non ritorna nulla)

Infine le Runnable non prendono niente e non ritornano nulle

(a differenza delle altre sta in java lang siccome era usato nei thread)

```
interface Runnable {
    void run();
}
```

11. Method reference && Functional interfaces

La keywork :: consente di chiamare un metodo di una classe (sia esso statico o dinamico), in generale abbiamo tre applicazioni possibili:

```
NomeClasse::nomeMetodo
oggetto::nomeMetodo
NomeClasse::new
```

- La prima riga riferisce solo metodi statici
- La seconda invece riferisce anche a quelli non statici (é una bad-practice riferire un metodo statico con oggetto::metodoStatico, si usi il primo modo)
- Il terzo invoca il costruttore.
  Attenzione, non li invoca

Se il nome del metodo, o del costruttore, coincide con la signature del metodo di un'interfaccia funzionale, avremo un'implementazione
dell'interfaccia che utilizza il metodo referenziato con l'operatore ::
che, a sua volta, si collega ai parametri di input ed outputdel metodo funzionale.
In sostanza come una lambda ma passi una funzione di un oggetto

Abbiamo detto che dobbiamo avere un'interfaccia funzionale...

11.1 Interfacce funzionali

Java 8 ha introdotto diverse novitá fra cui un metodo per programmare in stile "funzionale".
Un'interfaccia funzionale presenta un solo metodo.

In `java.util.function` l'interfaccia `Predicate<T>` é un chiaro esempio di interfaccia funzionale: ha solo il metodo `boolean test(T)` e infatti `Predicate` rappresenta un predicato (una funzione con un solo argomento che restituisce un valore booleano).

Esempio dell'utilizzo di predicate:

```
public class IsOdd implements Predicate<Integer>{
    @Override
    public boolean test(Integer n) {
        return n%2 ? false : true;
    }
}
```

12. Tread e design patterns

Sai giá cosa sono e se non lo sai ti consiglio di studiare Sistemi operativi 2, puoi trovare i miei appunti andando su: https://github.com/WAPEETY/appunti-SO2.

sono fatti dal Sistema Operativo non dal linguaggio di programmazione!

Per evitare il problema vecchissimo della sincronizzazione java ha inventato un construtto per gestire le sezioni critiche e si chiama `Syncronized()`.

12.1 Synchronized

In Java si può creare una critical sectiion attraverso la parola chiave `syncronized` con la seguente sintassi.

```
    synchronized (mutex) {
    // Code
}
```

- Nelle parentesi tonde riceve un oggetto che viene utilizzato come mutex.
- Viene poi aperto un blocco di istruzioni che contiene la critical section.
- Questo mutex può essere un oggetto qualsiasi, basta che sia condiviso - fra tutti i metodi che devono entrare nella sezione critica.
- La JVM utilizzerà questo oggetto per far eseguire il blocco da un thread alla volta.
- i mutex hanno un counter, se lo locki 18 volte (ovviamente sempre lo stesso owner) lo devi unlockare 18 volte e questo serve per la ricorsione che, altrimenti, non sarebbe possibile. Pensa ad un blocco del genere

```
foo(){
    synchronized(this){
        //qua faccio roba per vedere se é necessaria la ricorsione
        if(something){
            foo();
        }
    }
}
```

Se non avessero un counter per owner avremmo un thread bloccato ogni volta che `foo()` chiama se stessa!

Un oggetto qualsiasi in Java puó fare da mutex (semaforo) e la syncronized chiama una lock e unlock che sono segrete e non accessibili direttamente dal programmatore, cosí tu (che non sei capace a

programmare bene) non puoi fare bug.

Fra le tonde che metto?
Devo scegliere un oggetto che hanno tutti i thread che vuoi sincronizzare

syncronize ha anche uno zucchero sintattico:

```
private syncronized void foo(){
    System.out.println("funzione atomica");
}
```

che é come:

```
private void foo(){
    syncronized(this){
        System.out.println("funzione atomica");
    }
}
```

e ti assicura che tutta la funzione viene eseguita a una sola botta (termine tecnico per indicare che non ci saranno altri thread dentro quella funzione a rompere tutto nel frattempo che uno la esegue).

12.2 Modi per creare thread

12.2.1 Classe dedicata

Come già specificato prima, un modo per creare un thread è estendere la classe `Thread` per fare l'override del metodo `run`.

12.2.2 Implementando Runnable

Si puó implementare l'interfaccia runnable e quella classe sará usabile come un thread.

12.2.3 Oggetto `Thread` e method reference

In alternativa si può creare un thread istanziando un oggetto di classe `Thread` passando come parametro al costruttore un Runnable.

```
        public class Looper{
              public static void loop() {
                      int i = 0;
                      while (true) {
                              Thread.sleep(1000);
                              System.out.println("Loop " + i++);
                      }
```

```
                }
        }

        new Thread(Looper::loop).start();
```

12.2.4 Oggetto `Thread` e lambda

Ovviamente dato che `Thread` riceve un Runnable si può passare una lambda come parametro al costruttore.

```
new Thread(() -> {
     int i = 0;
     while (true) {
             Thread.sleep(1000);
       System.out.println("Loop " + i++);
     }
}).start();
```

12.3 Stop (DEPRECATO)

- I thread **non** si possono killare, devono fare seppuku (killarsi da soli)
- Anche se nella classe è presente il metodo `stop`, è deprecato e non va utilizzato perché può creare problemi.
    - Il problema del metodo `stop` ha a che fare con la sincronizzazione tra thread.
    - Se un thread `x` attraverso un semaforo mette in attesa un thread `y`, e poi `x` viene killato, il thread `y` non riprenderà più ad eseguire, causando un vero e proprio Deadlock.

12.4 Consumer/producer

Il Design pattern consumer/producer è uno dei più diffusi:
- Una componente produce dati.
- Un'altra componente li "consuma".

```
    public class ProducerConsumer {

      private List<Integer> l = new ArrayList<>();

      public static class Producer extends Thread {
              @Override
              public void run() {
                      Random rnd = new Random();
                      while (true) {
                              l.add(rnd.nextInt(100));
                      }
```

```
                }
        }

        public static class Consumer extends Thread {
                @Override
                public void run() {
                        while (true) {
                                int n = l.remove(0);
                                System.out.println(n);
                        }
                }
        }

        public static void main(String[] args) {
                Producer p = new Producer();
                Consumer c = new Consumer();

                p.start();
                c.start();

                try {
                        p.join();
                        c.join();
                } catch ( e) {
                        e.printStackTrace();
                }
        }
    }
```

- Questa versione si rompe subito per i seguenti problemi:
  - Innanzitutto se `Consumer` in qualsiasi momento si trova `l` vuoto, il thread crasha.
  - Inoltre, se `Consumer` crasha, `Producer` produce elementi all'infinito finché non finisce la memoria.
  - `Producer` e `Consumer` accedono a `l` senza gestirsi a vicenda, creando potenzialmente collisioni.

Nota! Gli altri design pattern li presento in breve giusto perché é utile saperli

12.5 Singleton

```
public static class Foo{
    //tramite questa variabile tengo traccia dell'istanza (siccome ne voglio solo una)
```

```
    private static Foo inst = null;
    public Foo getInstance(){
        if(inst == null)
            inst = new Foo(); //se non esiste la crea
        return inst;
    }
}
```

L'idea é quella di assicurare solo un oggetto (statico) e l'unica funzione disponibile (oltre il/i costruttore/i) ritorna una nuova instanza solo se non ne esiste giá una.

12.6 Factory pattern

```
public interface CarFactory{
    public Car makeCar();
}

interface Car{
    public String getType();
}

class SpecificCarFactory implements CarFactory{
    public Car makeCar()    {
        return new SpecificCar();
    }
}

class SpecificCar implements Car{
    public string getType(){
        return "My Specific Car Model";
    }
}

// Utilizzo
Factory f = new SpecificCarFactory();
Car c = f.makeCar();
System.out.println(c.getType());
```

L'idea alla base é quella di dividere l'implementazione dell'oggetto dalla creazione dello stesso, quindi come si puó vedere sopra si creano delle classi solo per "costruire" l'oggetto.
Questo permette di estendere solo la "costruzione" di un oggetto e non la sua implementazione (alla fine é un giro assurdo che in molti casi é abbreviabile con del semplice polimorfismo o dei fake constructor)

13. WildCards

In java esistono dei caratteri jolly che vengono usati in combinazione con i tipi parametrici. Questo perché Java non permette ereditarietá nei generics! (sempre colpa delle banche)

Quindi ad un tipo T generico non posso assegnare (nella stessa istanza) un tipo A e poi un tipo B che estende A, o almeno... non direttamente ma usando i Wildcards.

Esempi:

```
public class Wildcards{

    public static <T> T identity(T x){
        return x; //qui X deve essere dello stesso tipo
        //ES: non posso farmi ritornare un Animal e passare un Dog
    }

    public static Object identity2(Object x){
        return x; //questo funziona ma devo fare io il casting fuori (e non usa i generics)
    }

    //NON SI PUÓ FARE
    //public static ? identity3(? x){
    //  return x;
    //}
}
```

Differenze?

- la prima funzione mi permette di fare un nome al tipo e quindi essere sicuro che quello che mi ritornerá sará lo stesso elemento passato
- la seconda puoi passare tutto e ricevere tutto
- la terza non compila, infatti ci sono dei limiti all'uso della wildcard ? (altrimenti avresti un linguaggio senza tipi)

peró se il tipo é definito e parametrico posso farlo, ad esempio... posso passare una collection di cose e ricevere una collection di altre.

```
public static Collection<?> w1(Collection<?> x){
    //faccio cose
```

```
    return foo;
}
```

e questo é fondamentale, cosí non devo fare subsumption ad object (che é illegale)

Per mettere dei limiti ai tipi accettati esistono le wildcards con i bound.

13.1 Upper bound

`<? extends bound_type>`

Si utilizzano tra le parentesi angolari e il tipo in questo caso indica un upper bound ossia un vincolo superiore per il tipo che è possibile utilizzare durante la creazione di un tipo parametrizzato.

```
class A{}
class B extends A{}
class C extends B{}

public static <E extends B> void identity(E x){
    //in questa function posso passare obj di tipo C o B ma non di tipo A
}
```

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, dato un tipo `List<x>`, di assegnare legittimamente un oggetto del suo tipo a una variabile di tipo `List<? extends A>` se x è di tipo A o un sottotipo di A.

13.4 Lower bound

`<? super bound_type>`

Si utilizzano tra le parentesi angolari e il tipo in questo caso indica un lower bound ossia un vincolo inferiore per il tipo che è possibile utilizzare durante la creazione di un tipo parametrizzato.

```
class A{}
class B extends A{}
class C extends B{}

public static <E super B> void identity(E x){
    //in questa function posso passare obj di tipo A o B ma non di tipo C
}
```

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, dato un tipo `List<x>`, di assegnare legittimamente un oggetto del suo tipo a una variabile di tipo `List<? super T>` se x è di tipo B o un supertipo di B.

14. Sorting

Di seguito un esempio completo di utilizzo di ció che abbiamo fatto fino ad oggi.

```
public class Sorting{

    static<T extends Comparable<? super T>> void sort(List<T> l){
        Collections.sort(l);
    }

    public interface Shape extends Comparable{
        double area();
        double perimeter();

        Default int compareTo(Shape s){
            return (int)(this.area() - o.area())
        }
    }

    public static class Circle implements Shape{
        private double rad;

        public Circle(double r){
            this.rad = r;
        }

        public double area(){
            return rad * rad * Math.PI;
        }

        public double perimeter(){
            return Math.PI * 2. * rad;
        }

        @Override
        int compareTo(Shape s){
            if(s instanceof Circle){
                Circle c = (Circle) s;
                return (int) (area() - s.area());
            }
```

```
            return (int)(this.area() - o.area())
        }
    }

    public static class Rectangle implements Shape{
        private double b;
        private double h;

        public Rectangle(double b,double h){
            this.b = b;
            this.h = h;
        }

        public double area(){
            return b*h;
        }

        public double perimeter(){
            return b+h*2
        }
    }

    public static void main(String){
        List<Integer> l = List.of(6,87,-45,8787,46)
        Collection.sort(l);



        List<Shape> l2 = new ArrayList<>();

        l2.add(new Circle(4.));
        l2.add(new Rectangle(3.,9.));
        collection.sort(l2);
    }
}
```

```
static<T> void sort(List<T> l, Comparator<? super T> cmp){
    Collections.sort(l,cmp);
}

Collection.sort(l2,(Shape o1, Shape o2) -> (int) (o1.perimeter -
o2.perimeter))
```

E con questo esempio ci siamo levati tutto il levabile di Java.

15. C++ Intro

C++ é **ANCHE** ad oggetti ma non é prettamente ad oggetti.

Bjarne Straus.... (come si chiama) é un islandese e il suo nome sembra quello di un mobile IKEA.

cfront era il compilatore da c++ a c (transpliler)
Nel 2009 ha droppato i diritti et voiltá la community ha fatto un fottio di updates.

C++ é rotto dall'83

15.1 Moduli

i nuovi file per i moduli si chiamano `IXX` e si é creato un sistema tipo:

`export` per rendere il modulo visibile agli altri
`module <nomemodulo>` per creare un modulo con un nome

per importare:

```
import <nomemodulo>;

//all'utilizzo
nome_namespace::nomeclasse
```

tutta 'sta roba devastante esiste dal 2020.

15.2 costruttori strani

```
export module zoo;

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

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

```
    }
}
```

nel costruttore weight_(w) chiama il default constuctor di **int** per copia e quindi mette in weight_ il valore di w

`explicit` vicino il costruttore serve perché sennó potresti fare cose del tipo:

```
void fun(animal a){
    ...
}

//chiamare

fun(4); //e fará un animal con weight 4 *mindblow*
```

15.4 La keyword virtual

`virtual` serve se vuoi permettere l'override altrimenti C++ quando compila hardcoda le jump evitando di usare il sistema di "virtual table" e puntatori a metodo che usa java (e che rendono il tutto piú lento) quindi non potresti fare una classe figlia che fa l'override dei metodi siccome le chiamate a funzione non sarebbero "modificabili" a runtime.

Se levo virtual posso comunque fare overload sui figli ma non posso fare dynamic dispatching

15.5 Appunti sugli oggetti i C++

gli oggetti in c++ possono stare sullo stack o sullo heap

```
    //sullo stack
    animal a(4);

    //sullo heap
    animal* b = new animal(4);
```

```
//extends
class dog : public animal{ //public etc... permette di far sapere chi puó subsumerti
    protected:
        bool has_pedigree;

    public:                     //animal(w) invoca il costruttore di animal
```

```
        dog(int w,bool has_ped) : animal(w), has_pedigree(has_ped) {}
}
```

per estendere un'altra classe si usa la keyword `:` e a differenza di Java dove si puó fare subsumption di qualsiasi tipo `reference` fino ad Object qui puoi decidere tramite la keyword `public` prima della classe che vuoi estendere chi poi effettivamente potrá sapere che sei una classe figlia di `animal`.

16. C++ Template

Qui apriamo il vaso di pandora!

C++ é un liguaggio compilato e non interpretato e il suo template system é un sistema generativo guidato di codice, **non é polimorfismo!**

Che significa? che i template spariscono a tempo di compilazione e vengono create tante varianti di quel codice templatizzato quante ne servono al programma per funzionare, é letteralmente un modo per far scrivere codice al compilatore!

```
template <class T>
T sum2(T a, T b){
    return a + b;
}
```

---

tutto si puó fare... basta definire
~Alvise Spanó

16.1 value_type

```
template <class C>
C::value_type sum(const C& v){
    if(v.size() > 0){
        C::value_type r(v[0]);
        for(size_t i = 0;i < v.size(); ++i){
            C::reference = x(v[i]);
            r+= x;
        }
        return r;
    }
    return C::value_type();
}

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

in c++ puoi avere tipi dentro tipi e quindi il template system sará a conoscenza del tipo che avrá quella "C" generica che abbiamo messa e creerá tante sum quante ne servono con il tipo giustoa compile time!

17. C++ polimorfismo

`value_type` é un typedef dentro `vector`

In C++ l'operatore di assegnamento é una chiamata a funzione, la funzione di assegnamento del left value

la subsumption in C++ puó avvenire by reference o by pointer NO by value

```
animal a(69);
dog b(10);
a = b;
animal c = b; //questo non é un rebinding, é proprio un nuovo object
```

IMPORTANTE

`c.eat(&c)` é fatto su animal, costruendo e non rebindando perdo info (non so che era un cane)

per fare alla Java devo lavorare di pointer

```
animal* a = new animal(69);
dog* b = new animal(10);
a=b;
animal* c = b;
c.eat(c); //qua invece é fatto su dog

//mi devo ricordare le delete
delete a;
delete b;
```

18. C++ Lambdas

C++ 11 ha portato le lambdas (tardi) che
non hanno tipo (lo hanno ma é capito in automatico)

```
// questa versione della sum funziona sugli iteratori, naturalmente
templatizzati
// tuttavia assume l'esistanza del member type value_type, quindi non
dovrebbe funzionare con i pointer a meno di usare i traits
// e funziona da c++ 11 in poi
    template <class InputIterator>
    auto sum(InputIterator first, InputIterator last)
    {

        while (first != last)
        {
            r += *first++;
        }
        return r;
    }
```

```
template <class InputIterator, class BinFun>
auto sum(
        InputIterator first,
        InputIterator last,
        Binfun f;
         ){
                auto r(*first);
                while(first != last){
                    r = f(r, *first++);
                }
                return r;
}
```

quindi questo accetta oggetti con operator tonde, function pointer e lambdas

---

se volevate le cose facili andavate ad economia ~Spanó

18.1 Test vari e closure

Adesso iniziamo a spaccarci le ossa veramente con le lambda!

```
export void test()
    {
        // proviamo un po' di lambda
        {
            auto f1 = [](int x) { return x + 1; };      // da int a int
            auto f2 = [](auto x) { return x + 1; };     // con auto sul
lambda parametro
            auto f3 = [](auto x) -> int { return x + 1; };      // con
annotazione esplicita del tipo di ritorno ed auto nel lambda parametro
            auto f4 = [](int x) -> int { return x + 1; };       // con
annotazione esplicita sia del tipo del lambda parametro che del tipo di
ritorno
        }

        // altri esempi con reference
        {
            auto f1 = [](const int& x) { return x + 1; };       // con un
const int& come lambda parametro
            auto f2 = [](const auto& x) { return x + 1; };      // auto può
essere usato insieme a const e reference: il compilatore non inferisce mai &
e const con auto, inferisce solo il tipo principale
            auto f3 = [](auto& x) { x++; };
// come reference non-const
        }
```

```
        // esempi di capture: si chiamano capture le variabili catturate
dalla chiusura della lambda
        // c++ permette di customizzare il comportamento delle capture in
maniera molto fine
        {
            int k = 5;
            vector<int> v{ 1, 2, 3, 4, 5 };
            auto f1 = [=](int x) { return x + v[0] + k; };      // v e k
sono catturate per COPIA nella chiusura della lambda
            auto f2 = [&](int x) { return x + v[0] + k; };      // v e k
sono catturate per REFERENCE nella chiusura della lambda
            auto f3 = [=, &v](int x) { return x + v[0] + k; };  // tutto per
copia (cioè solo k, nel nostro caso) eccetto v per reference
            auto f4 = [&, k](int x) { return x + v[0] + k; };   // tutto per
reference (cioè solo v, nel nostro caso) eccetto k per copia
            auto f5 = [a = v, b = k](int x) { return x + a[0] + b; };   //
tutto per copia con rebinding dei nomi: v si chiama a e k si chiama b
            auto f6 = [&a = v, b = k](int x) { return x + a[0] + b; };  // v
```

```
si chiama a ed è per reference; k si chiama b ed è per copia
        }
    }
```

19. Smart Pointer

Sarebbe comodo sopravvivere alla deallocazione di memoria, per questo qualcuno si é inventato gli smart pointer!

```
class C{
    private:
        int* p;
    public:
        C(int* p_) : p(p_) {} //copia il contenuto di *(quindi l'indirizzo)
        int* get() const { return p; }
}

export void test(){
    int* a = new int(23);
    C c(a);
    int* b = c.get();
}
```

é il solito problema dell'ownership dei pointer che si risolve o con i garbage collector oppure...

Smart Pointer
Sono meglio di niente, non sono perfetti ma ti aiutano sicuramente

```
class C{
    private:
        smart_ptr<int> p;
    public:
        C(int* p_) : p(p_) {} //copia il contenuto di *(quindi l'indirizzo)
        smart_ptr<int> get() const { return p; }
}

export void test(){
    smart_ptr<int> a = new int(23);
    C c(a);
    smart_ptr<int> b = c.get();
}
```

ovviamente va implementato, e lo implementiamo cosí:

```
template <class T>
class smart_ptr{
```

```
    private:
        T* pt;
        size_t* cnt;
        ptrdiff_t offset;

        dec(){
            ~~(*cnt);
            if(*cnt == 0)
            {
                if(is_array){
                    delete[] pt;
                }
                else {delete pt;}
                delete cnt;
            }
        }

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

        smart_ptr(T* pt_, ptrdiff_t offset_, size_t* cnt_, bool is_array_) :
pt(pt_), offset(offset_), cnt(cnt_), is_array(is_array_){ inc(); }
    public:
        smart_ptr(T* p, is_array_ = false) : pt(p), cnt(new
int(1),is_array(is_array_) ){}
        smart_ptr(const smart_ptr<T>& p) : pt(p.pt), cnt(p.cnt){
            inc();
        }

        ~smart_ptr(){
            dec();
        }

        smart_ptr<T>& operator=(const smart_ptr<T>& p){
            if(pt == p.pt){ //se sono io stesso non faccio nulla
                return *this;
            }
            dec(); //decremento quello vecchio
            ++(*p.cnt); //incremento quello nuovo
            this->pt = p.pt;
            this->cnt = p.cnt;
            this->is_array = p.is_array
```

```
        return *this;
    }

    T& operator*(){
        return pt[offest];
    }

    const T& operator*() const{
        return pt[offest];
    }

    bool operator==(const smart_ptr<T>& p) const{
        return pt == p.pt && offset == p.offset;
    }

    bool operator==(const smart_ptr<T>& p) const{
        return !(*this == p); // cosí chiama ==
    }

    operator T*(){
        return pt[offest];
    }

    operator const T*() const{
        return pt[offest];
    }

    smart_ptr<T> operator+(ptrdiff_t d) const{
        return smart_ptr<T>(pt, offset + d, cnt);
    }

    smart_ptr<T> operator-(ptrdiff_t d) const{
        return *this + (-d);
    }

    smart_ptr<T>& operator+=(ptrdiff_t d){
        offset+= d;
        return *this;
    }

    smart_ptr<T>& operator-=(ptrdiff_t d){
        return *this += -d;
```

```
        }

        smart_ptr<T>& operator++(){
            return *this += 1;
        }

        smart_ptr<T>& operator++(int){
            smart_ptr<T> tmp(*this);
            ++(*this);
            return r;
        }

        smart_ptr<T>& operator--(){
            return *this -= 1;
        }

        smart_ptr<T>& operator--(int){
            return *this -= 1;
        }

        T* operator->(){
            return pt + offset;
        }

        const T* operator->(){
            return pt + offset;
        }

        T& operator[](size_t i) {
            return pt[offset+i]
        }

        const T& operator[](size_t i) const {
            return pt[offset+i]
        }
}
```

PS: esistono nella standard library ma si chiamano shared pointer (perché sono pure thread safe)

20. Pattern per evitare ripetizione

Prendendo in esempio il codice degli smart pointer

```
T& operator*(){
    return const_cast<T&>(*std::as_const(*this))
}

operator const T*() const{
            return pt[offest];
}
```

const_cast puó togliere o mettere il const

gli passi un'espressione `*std::as_const(*this)`:

- c'é un dereference di una chiamata a funzione che prende il dereference del this come parametro

questo fa si che all'inizio aggiungo un const (con as_const) e lo chiamo, questo fa chiamare la funzione con la versione const e poi gli levo il const

Altro trick:

sempre con la classe `smart_ptr`

```
template <class T, size_t Len = 1>
    class smart_ptr
    {
        //... roba giá scritta

        void dec(){
            ~~(*cnt)
            if(*cnt == 0)
            {
                if constesxpr (Len >= 1) delete[] pt
                else delete pt

                delete cnt;
            }
        }
```

```
        //...

        smart_ptr<T, Len>& operator=(const smart_ptr<T, Len>& p)
    }
```

in questo modo compilerá la versione giusta che chiamerá la delete giusta a seconda se abbiamo un pointer singolo o un "array di pointer"

sarebbe bello farglielo fare in automatico... e infatti

```
template <class Ty, size_t Len = (std::is_array_v<Ty> ? std::extent_v<Ty,0>
: 1)>

private:
    using T = std::remove_extent_t<Ty>;
```

fra tonde é un'espressione valutabile a compile time che se é un array chiama `extent_v<T, 0>` -> ovvero:

di questo coso T, quanto é lunga la dimensione 0? (potrei avere anche matrici o peggio array a N dimensioni)