Contents

JAVA

Object Oriented Programming

Main

```java
public class Main {
    public static void main(String[] args) { }
}
```

Keywords

- `final`: attributo immutabile (**deve** essere inizializzato nel costruttore)
- `final`: metodo che non può essere sovrascritto
- `static`: il metodo viene chiamato senza instanziare un oggetto, ma direttamente sulla classe
- di default tutto inizializzato a `0` o `null`
- `switch`:

```java
switch(x) {
    case 'A' -> { }
    default -> { }
}
```

- `abstract`:

```java
public abstract class Poly {
    public abstract void draw();
}
public quad extends Poly {
    @Override public void draw() { }
}
```

- `interfaces`:

```java
public Interface Printable {
    // `abstract` implicito
    public void print();
}
public quad implements Printable {
    @Override public void print() { }
}
```

Visibilità

- `public`: visibile a tutti
- `protected`: visibile alle sottoclassi e nel package
- "friendly" by default: visibile nel package
- `private`: visibile solo alla classe stessa (non nelle sottoclassi)

Object

- Ogni classe è figlia di `Object`, che definisce i seguenti metodi:

```
public String toString() {
    return getClass().getName() + "@"
    + Integer.toHexString(hashCode());
}
public final boolean equals() {
    return (this == obj);
}
// run-time class of the object
public final Class<?> getClass();
public int hashCode();
```

- `equals` è meglio ridefinirla su attributi `final`

```
public final boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Foo f)) return false;
    // `o` viene castato a `f` alla linea sopra
    return bar == f.bar;
}
```

Exceptions

```
public class EmptyListException extends Exception {
    public EmptyListException(String message)
        { super(message); }
}
```

- uncheck exceptions: no compile-time checking (`throws` non necessario)
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - IllegalArgumentException
  - ArithmeticException

Throws

```
public void f(int x) throws negIntException {
    if (x < 0) throws new negIntException();
}
```

Try Catch Finally

```
try { }
catch (ExceptionType e) { }
// eseguito indifferente dall'esito, prima del return
finally { }
```

Generics

```
public static <T> T getFirst(List<T> list) { }
public static <T extends Comparable<T>>
    T getMax(List<T> list) { }
```

Concurrency

- solo gli oggetti sull'Heap possono usufruire di `synchronized`
- per avere un Deadlock, è sufficiente che un thread sia per sempre bloccato
  - **non** che una risorsa condivisa sia per sempre non raggiungibile
- se ci sono setters, è meglio usare `synchronized` anche per i getters
- i lock è meglio avvenago su attributi `final`
  - altrimenti il lock (l'oggetto) può essere sostituito
- a volte capita di dover mettere tutte le funzioni come `synchronized`
  ad esempio se si ha bisogno di un `||` su lock diversi

Synchronized

```
public void g() {
    synchronized(x) {
        x.add(1);
        /* don't */ x.notify() // only one wakes up
        x.notifyAll();
    }
}
public void f() throws InterruptedException {
    synchronized(x) {
        while(x.isEmpty()) x.wait();
    }
}
public void h() {
    synchronized(x) {
        while(x.isEmpty()) {
            try (x.wait())
            catch(InterruptedException ex)
            { ex.printStackTrace(); }
        }
    }
}
```

Oppure si può anche rendere una lista sincronizzata

```
myList = Collections.synchronizedList(myList)
```

Read-Write Locks

accessi simultanei in lettura non vanno in conflitto

```
ReadWriteLock x = new ReentrantReadWriteLock()
// blocca read and write
x.writeLock().lock()
x.writeLock().unlock()
// blocca solo read
x.readLock().lock()
x.readLock().unlock()
```

Atomic

```
AtomicInteger x;
x.get() -> int
x.addAndGet() -> int
x.set();
```

Threads

```
publid void f() throws InterruptedException {
    Thread t = new Thread(Runnable);
    t.start();
    t.join();
}
```

o equivalentemente

```
publid void f() throws InterruptedException {
    Thread t = new Thread( public void run() {...} );
    t.start();
    t.join();
}
```

oppure ancora

```
class Worker implements Runnable { ... }

Worker w = new Worker();
new Thread(w).start();
```

Thread pool

```
public void f() throws ExecutionException { };
ExecutorService ex = Executors
    .newFixedThreadPool(int n);
// returns `null` in the `Future<?>`
ex.submit(Runnable action);
// prima completa i task
ex.shutdown();
// quanTo aspettare al massimo per shutdown
// bisogna prima chiamare `shutdown()`
while (!ex.awaitTermination(10, TimeUnit.SECONDS));
// non completa i task in sospeso
ex.shutdownNow();
```

Futures

```
Future<T> x = ex.submit(Runnable action, T result);
Future<T> x = ex.submit(Callable<T> action);
// aspetta finchè non lo ottiene
x.get(); // throws `ExecutionException` ...
Future<T> x = ex.submit(Callable<T> action);
FutureTask task = new FutureTask(Callable);
Thread t = new Thread(task);
task.get();
```

Interfaces

Comparator<T> e Comparable<T>

```
public class X implements Comparator<T> {
    // <0(less), 0(equal), >0(greater)
    public int compare(T a, T b) { }
}
public class Y implements Comparable<T> {
    // <0(less), 0(equal), >0(greater)
    public int compareTo(T other) { }
}
```

Iterator<T> e Iterable<T>

```
public class X implements Iterator<T> {
    public boolean hasNext() { }
    public T next() throws NoSuchElementException { }
    // from the underlying collection (not compulsory)
    public void remove()
        throws UnsupportedOperandException;
    // (not compulsory)
    public void forEachReaining(Consumer<T> c);
}
public class Y implements Iterable<T> {
    @Override public Iterator<T> iterator() {
        return new YIterator(this);
    }
    private class YIterator implements Iterator<T> {
        public YIterator(Pair<T> pair) { }
        public boolean hasNext() { }
        public T next() { }
    }
    // (not compulsory)
    public void forEach(Consumer<T> c);
}
```

Syntactic Sugar

```
for (Integer s : l1) { System.out.println(s); }
```

becomes

```
Iterator<Integer> it = l1.iterator();
while (it.hasNext()) {
    Integer s = it.next();
    System.out.println(s);
}
```

Function <T, R>

```
.apply(T arg) -> R
.andThen(Function<R, V> f) -> Function<R, V>
// come `andThen()` ma in ordine opposto
.compose(Function<V, T> f) -> Function<V, R>
.identity(T arg) -> T
public void foo(Function<Product, Integer> f)
    { f.apply(x); }
```

UnaryOperator<T>

- `BiFunction<T, T>`

BiFunction<T, U, R>

```
.apply(T arg, U arg) -> R
.andThen(Function<R, V> f) -> BiFunction<T, U, V>
```

BinaryOperator<T>

- `BiFunction<T, T, T>`

Consumer<T>

```
.accept(T) // esegue la funzione
.andThen(Consumer<T> c) -> Consumer<T>
```

Supplier<T>

```
.get() -> T
```

Predicate<T>

```
.test(T t) -> boolean
.and(Predicate o) -> boolean
.or(Predicate o) -> boolean
.negate() -> boolean
```

Runnable

```
.run() -> void
```

Callable<V>

```
.call() -> V // throws Exception
```

Functional Programming

```
public static Predicate<String> pred(String prefix) {
    return s -> s.startsWith(previs);
}
l.stream().filter(pred("X"))
```

- le seguenti formulazioni sono equivalenti:
  (se non per l'uso di `this`)

```
Function<Integer x, String y> g = new Function<>() {
    @Override public String y apply(Integer x)
        { return x.toString() }
}
Function<Integer x, String y> f = (x) -> x.toString();
```

- **Solo** nel caso una Lamda sia definita come nel primo caso,
  - `this` si riferisce alla lambda stessa. (si potrebbe ad esempio usare `this.apply()`)
  - `X.this` si riferisce alla classe `X` esterna.
- Nel secondo caso, invece
  - `this` si riferisce a `X`
- Nella Lambda si possono catturare solo variabili:
  - `final`
  - "effectively final": il compilatore è in grado di inferire che sia `final`
  - i puntatori possono comunque cambiare il contenuto
  - nei metodi, siccome `this` è un puntatore, gli attributi possono cambiare
- Syntactic sugar
  - `.map(Book::author)` diventa uno tra:
    - `.map(x -> x.author())` (x è di tipo Book)
    - `.map(x -> Book.author(x))`
  - se entrambi definiti compiler error

Stream<T>

- Static

```
Stream.of(T value) -> Stream<T>
Stream.generate(Supplier<T> s) -> Stream<T>
Stream.concat(Stream<T> x, Stream<T> y) -> Stream<T>
```

- General

```
.parallel() -> Stream<T>
// opposite of `parallel()`
.sequential() -> Stream<T>
.iterator() -> Iterator<T>
.findFirst() -> Optional<T>
.findAny() -> Optional<T>
.distinct() -> Stream<T>
.count() -> int
.toArray() -> T[]
.skip(long n) -> Stream<T>
.limit(long n) -> Stream<T>
```

- Collectors

```
.reduce(BinaryOperator<T> accumulator) -> U
.reduce(T identity, BinaryOperator<T> acc) -> T
    .reduce(0, Integer::sum)
.reduce(U identity, BiFunction<U, T, U>,
        BinaryOperator<U> accumulator) -> U
// `accumulator` è all'interno dello stesso thread
// `combiner` tra diversi threads `parallel()`
.collect(Suppiler<R> supplier, BiConsumer<R, T> acc,
       BiConsumer<R,R> combiner) -> R
  .collect(ArrayList::new, ArrayList::add,
           arrayList::AddAll)
  .collect(StringBuilder::new, (x, y) -> x.append(y),
      (x, y) -> x.append(", ").append(y)).toString()
.collect(Collector<R> collector) -> R
  .collect(Collectors.toList())
  .collect(Collectors.toSet())
  .collect(Collectors.joining(", "))
  .collect(Collectors.groupingBy(w -> w,
    Collectors.counting())
  // throws if duplicate
  .collect(Collectors.toMap(Function<T, R>
    keyMapper, Function<T, V> valueMapper)
  .collect(Collectors.toMap(Function<T, R>
    keyMapper, Function<T, V> valueMapper,
    BinaryOperator<v> merger)
  .collect(Collectors.toMap(x -> x.foo(),
    x -> x.bar(), (a, b) -> a + b, HashMap::new))
.toList() -> List<T>
```

- Consumer

```
.forEach(Consumer<T>)
// come `forEach` ma non consuma la stream
.peek(Consumer<T>) -> Stream<T>
```

- Function

```
.map(Function<T, R>) -> Stream<R>
.mapToInt(ToIntFunction<T>) -> IntStream
.mapToLong(ToLongFunction<T>) -> LongStream
.mapToDouble(ToDoubleFunction<T>) -> DoubleStream
.flatMap(Function<T, Stream<R>>) -> Stream<R>
```

```
.flatMapToInt(ToIntFunction<T>) -> IntStream
.flatMapToLong(ToLongFunction<T>) -> LongStream
.flatMapToDouble(ToDoubleFunction<T>) -> DoubleStream
```

- Predicate

```
.filter(Predicate<T>) -> Stream<T>
.takeWhile(Predicate<T>) -> Stream<T>
.dropWhile(Predicate<T>) -> Stream<T>
.anyMatch(Predicate<T>) -> boolean
.allMatch(Predicate<T>) -> boolean
.noneMatch(Predicate<T>) -> boolean
```

- Comparator

```
.sorted() -> Stream<T>
.sorted(Comparator<T>) -> Stream<T>
.min(Comparator<T>) -> Optional<T>
.max(Comparator<T>) -> Optional<T>
    .max((a, b) -> Float.compare(a.foo(), b.foo()))
        -> Optional<T>
    .max(Comparator.naturalOrder()) -> Optional<T>
```

IntStream (DoubleStream e LongStream)

- Static

```
IntStream.iterate(int seed, IntUnaryOperator f)
    IntStream.iterate(0, x -> x+1).limit(7)
```

- General

```
.range(int startIncl, int endExcl) -> IntStream
.boxed() -> Stream<Integer>
.sum() -> int
.max() -> Optional<Integer>
.min() -> Optional<Integer>
.average() -> Optional<Double>
.asIntStream() -> IntStream
.asLongStream() -> LongStream
.asDoubleStream() -> DoubleStream
```

- Function

```
.mapToObj(Function<T, R>) -> Stream<R>
.map(ToIntFunction<T>) -> IntStream
.mapToLong(ToLongFunction<T>) -> LongStrea-m
.mapToDouble(ToDoubleFunction<T>) -> DoubleStream
```

- General

```
.stream() -> Stream<T>
.isEmpty() -> boolean
.isPresent() -> boolean
.ifPresent(Consumer<T> action, Runnable emptyAction)
.isPresentOrElse(Consumer<T> action, Runnable mtA)
.orElse(T value) -> T
.orElseGet(Supplier<T>) -> T
.orElseThrow(T value) -> T
.get() -> T // error if `isEmpty()`
.filter(Predicate<T>) -> Optional<T>
```

- Function

```
.map(Function<T, R>) -> Optional<R>
.flatMap(Function<T, Optional<U>>) -> Optional<U>
```

Data Structures

Enums

```
public enum Type {
    A("Arancia") // constructor
    B("Bibbi")
}
Type x = Type.A;
```

```
T.values() -> T[]
.ordinal() -> int
.compareTo(T e) -> int
```

Array T[]

- **non** esiste `new`

```
int[] arr = new int[20];
int[] arr = {1, 2, 3, 4, 5};
arr = (T[]) new Object[10];
```

```
.length -> int // non .length()
// solo il primo livello, non ricorsivamente
.clone() -> T[]
```

List<T>

- New

```
List<T> l = new LinkedList<>()
List<T> l = new ArrayList<>()
List<T> l = new ArrayList<>(Collection<T> other)
List<double> l = Arrays.asList(1.0, 2.0)
.subList(int from, int to) -> List<T>
```

- General

```
// `o` must must be of type `List<_>`,
// and contain the same ordered elements
.equals(Object) -> boolean
.size() -> int
.forEach(Consumer<T>)
.sort(Comparator<T>)
.clear()
```

- Setters

```
// returns `true` if the set changed;
.addAll(Collection<T> c) -> boolean
// shifts the remaining part
// of the original list to the left
.addAll(int idx, Collection<T> c) -> boolean
.add(T value)
.add(in idx, T value)
.replaceAll(UnaryOperator<T>)
```

- Getters

```
.indexOf(Object o) -> int
.lastIndexOf(Object o) -> int
.get(int idx) -> T
.contains(T value) -> boolean
.containsAll(Collection<Object> c) -> boolean
.isEmpty() -> boolean
```

- Transformer

```
.toArray() -> T[]
.stream() -> Stream<T>
.iterator() -> Iterator<T>
    .hasNext()
    .next()
    .nextIndex()
    .hasPrevious()
    .previous()
    .previousIndex()
    // replace the last processed element
    // from the underlying list
    .set(T value)
    // remove the last processed element
    // from the underlying list
    .remove()
    // after the last processed element
    // in the underlying list
    .add(T value)
```

- Remove

```
.remove(Object O)
.removeIf(Predicate<T> filter)
.removeAll(Collection<Object>) -> boolean
.retianAll(Collection<Object>) -> boolean
```

Queue<T>

- Setters

```
.offer(T value) -> boolean
// throws IllegalStateException
.add(T value) -> boolean
.addAll(Collection<T>) -> boolean // throws ...
```

- Getters

```
.peek() -> T // or null
.element() -> T // or NoSuchElementException
.contains(T value) -> boolean
.containsAll(Collection<Object>) -> boolean
```

- Remove

```
// retrives and removes
.poll() -> T // or null
.remove() -> T // throws NoSuchElementException
```

Deque<T>

```
Deque<T> extends Queue<T>
```

- come Queue, ma `...First()` and `...Last()` (tranne per `.element()`)

Set<T>

- New

```
Set<T> s = new HashSet<>()
Set<T> s = new HashSet<>(hs)
Set<T> l = new HashSet<>(Collection<T> other)
```

- General

```
// `o` must be Set<_>,
// and contain the same elements
.equals(Object) -> boolean
.size() -> int
.clear()
.forEach(Consumer<T>)
```

- Transformer

```
.toArray() -> T[]
.stream() -> Stream<T>
.iterator() -> Iterator<T>
```

- Setters

```
// returns `true` if the set changed
.add(T value) -> boolean
.addAll(Collection<T>) -> boolean
```

- Getters

```
.contains(T value) -> boolean
.containsAll(Collection<Object>) -> boolean
.isEmpty() -> boolean
```

- Remove

```
.remove(Object) -> boolean
.removeif(Predicate<T>) -> boolean
.removeAll(Collecionts<Object>) -> boolean
.retainAll(Collecionts<Object>) -> boolean
```

Map<K, V>

- New

```
Map<K, V> m = new HashMap<>()
Map<T> l = new HashMap<>(Collection<T> other)
```

- General

```
// `o` must be of type Map<_, _>,
// and contain the same keys with the same values
.equals(Object o) -> boolean
.size() -> int
.clear()
.forEach(BiConsumer<K, V> action)
```

- Setters

```
.put(K key, V value) -> V // previous or null
.putIfAbsent(K key, V value) -> V // previous or null
.putAll(Map<K, V> m) // replaces
```

- Getters

```
.get(K key) -> V // `null` se `!containsKey(K)`
.getOrDefault(K key, V value) -> V
.entrySet() -> Set<Entry<K, V>>
.keySet() -> Set<V>
.values() -> Collection<V>
.containsKey(K key) -> boolean
.containsValue(V value) -> boolean
```

- Remove

```
.remove(K key) -> V // removed value
.remove(K key, V value) -> boolean
```

- Replace

```
.replace(K key, V value) -> V
.replace(K key, V oldValue, V newValue) -> boolean
.replaceAll(BiFunction<K, V, V>) -> Set<V>
```

EntrySet<K, V>

- Getters

```
.getValue() -> V
.setValue() -> V // previous value or `null`
.getKey() -> K
```

Optional<T>

- Static

```
Optional.of(T value) -> Optional<T>
// `of()`, but empty if `value == null`
Optional.ofNullable(T value) -> Optional<T>
Optional.empty() -> Optional<T>
```

- Getters

```
.isPresent() -> boolean
.isEmpty() -> boolean
.get() -> T // throws NoSuchElementExceptio
.orElse(T val) -> T
.orElseThrow() -> T // throws ...
.orElseThrow(Supplier<> exceptionSupl) -> T
.orElseGet(Supplier<T>) -> T
```

- Mapper

```
.or(Supplier<Optional<T>>) -> Optional<T>
.map(Function<T, U>) -> Optional<U>
.flatMap(Function<T, Optional<U>>) -> Optional<U>
```

- Consumer

```
.ifPresent(Consumer<T>)
.ifPresentOrElse(Consumer<T>, Runnable)
```

- Predicate

```
// empty if `false`
.filter(Predicate<T>) -> Optional<T>
```

String

- Setter

```
.replace(String regex, String) -> String
.replaceFirst(String regex, String) -> String
.concat(String) -> String
.toUpperCase(String) -> String
.toLowerCase(String) -> String
```

- Getters

```
.length() -> int
.isEmpty() -> boolean
.contains(char) -> boolean
.charAt(int idx) -> char
.lastIndexOf(int ch) -> int // `-1` if not found
.lastIndexOf(String) -> int // `-1` if not found
.matches(String regex) -> boolean
.indexOf(String) -> int
.lastIndexOf(String) -> int
.compareTo(String) -> int
.compareToIgnoreCase(String) -> int
.lines() -> Stream<String>
.chars() -> IntStream
```

Math

- `int` si auto-casta a `double`, ma non viceversa

```
Math.max(int x, int y);
Math.min(int x, int y);
Math.abs(double x);
Math.ceil(double x);
Math.floor(double x);
System.out.println(double);  // .0
Integer.compare(int x, int y);
Double.sum(int x, int y);
Integer.MIN_VALUE
Double.MAX_VALUE
```

IO

- new

```
Scanner in = new Scanner(System.in);
Scanner in = new Scanner(new File("file/path");
PrintStream out = new PrintStream(System.out);
PrintStream out = new PrintStream(new File("file");
```

- read

```
.hasNext() -> boolean
.next() -> String
.nextLine() -> String
.nextInt() -> int
.hasNextInt() -> boolean
.close() -> void
```

- write

```
.print(Object o)
.println(Object o)
.printf(String format, Object... args)
```

Test

- **statement coverage**: sollecita ogni statement
- **decision (edge) coverage**: per ogni branch, prova sia true che false
  - va preso anche il ramo `else` supper "non esista"
  - "non posso coprire tutte le dicisioni, siccome..."
- **condition coverage**: per ogni branch, provo ogni combinazione possibile
  - `x && y` => 0_, 10, 11
  - `x || y` => 1_, 01, 00
- **path coverage**: ogni possibile cammino, da inzio a fine, nella funzione

Ereditarietà

1) ricopia i metodi: **attenzione** a `this` e `super` nei metodi ereditati
   - `this` è ricopiato con un cast alla classe da cui viene ereditato
   - `super` è riferito alla superclasse da cui viene ereditato
   - `this.m()` cerca il metodo `m()` più specializzato
     - anche nelle sottoclassi
     - anche nei metodi ereditati
   - `super.m()` cerca il metodo `m()` più specializzato nelle superclassi
     - si ferma a quella appena sopra al chiamante
     - se ereditato, `super` va considerato rispetto alla classe in cui il metodo è definito
2) scegli il metodo statico:
   - considera il tipo statico (più specializzato) del chiamante
   - considera il tipo statico (più specializzato) degli argomenti
   - (cast come `int` -> `float` avvengono solo dopo che non si trovano match con `int`)
3) guardando il tipo dinamico, scegli l'Override più specializzato

JML

Sezioni

- I metodi `ground` sono il più piccolo insieme di metodi puri da cui definire gli altri
- `@requires <expr>` precondizioni che devono essere verificate per un corretto funzionamento: sono ciò che le eccezioni non coprono
  - di solito param `!= null` e ...
- `@ensures <expr>` postcondizioni verificate al termine, se `@requires` rispettate
  - di solito `\result != null` e ...
  - assicurati il risultato non contenga elementi esterni inseriti a caso
  - ricorda di `@ensures !<@signals> && <requires> && ...`
- `@signals(exceptionName e) <expr>` cosa è vero quando si verifica l'eccezione
  - ricorda di mettere "nothing changes"
- `@private invariant`
  - la **representation invarian** descrive quando uno stato concreto è valido
  - la **abstract function** descrive come uno stato concreto corrisponde a uno stato astratto
- `@assignable x` (`T x` è un parametro in ingresso) indica che `x` può essere modificato
- `@assignable v[*]` (`T[] v` è un parametro in ingresso) idem
- `@spec_public <type> oat;` (dove `oat` è l'Oggetto Astratto Tipico) serve per descrivere stati invisibili all'utilizzatore
  - non corrisponde necessariamente all'implementazione
  - esempio: una stack descritto tramite una lista

Sintassi

- `==>`, `<==`, `<==>`, `<=!=>`, `||`, `&&`
- `\this`, `(\old f())`, `\result`
- `(* comment *)`

Siano:

- `<varDecl>` = `T varName`
- `<domain>` = `T -> boolean`
- `<expr>` = `T -> boolean`
- `<toNum>` = `T -> int`
- `<toNum>` = `T -> double`

allora:

- `(\forall <varDecl>; <domain>; <expr>)`
  - a volte si vuole predicare su tutti gli elementi, non solo su quelli dell'insieme
- `(\exists <varDecl>; <domain>; <expr>)`
- `(\numof <varDecl>; <domain>; <expr>)`
- `(\min <varDecl>; <domain>; <toNum>)`
- `(\max <varDecl>; <domain>; <toNum>)`
- `(\sum <varDecl>; <domain>; <toNum>)`
- `(\product <varDecl>; <domain>; <toNum>)`

Predicati e funzioni speciali

- posso definire predicati: `pred1(x) <==> x == y`
- posso definire funzioni: `f1(x) == x.size()`
  - e queste possono anche essere ricorsive

Estensioni

- **Estensione pura** che non modifica la specifica dei metodi ereditati
  - la sottoclasse può usare sono metodi `public` o `protected`

Principio di sostituzione di Liskov

Una estensione è valida se: - **Signature**: garantita dal compiler - **Metodi**: eredita la specifica con al più le seguenti modifiche: - pre-condizioni meno restrittive (`X || Y` e `Y ==> X`) - post-condizioni più restrittive (`X && Y`) - le eccezioni si possono togliere solo se - dipendenti da uno stato non più raggiungibile - opzionali - `@also requires pre_sub` becomes `@requires pre_super || pre_sub` - `@also ensures post_sub` becomes `@ensures (\old(pre_super) ==> post_super) && (\old(pre_sub) ==> post_sub)` - **Proprietà**: - `@public invariant` preservati - proprietà evolutive: - non devono evolvere identicamente - ma gli stati raggiungibili nella sotto-classe lo devono
anche essere utlizzando solo metodi della super-classe

Tips

- solo metodi modificator (non `@pure`) bisogna mettere cosa cambia e cosa non
  mentre se sono `@pure` allora non è necessario specificare che nulla cambi
- no duplicati: `x.stream().distinct().count() == x.size()`
- assicurati non vengano chiamate exceptions dai metodi chiamati
- attenzione ai **duplicati** nelle liste
- "nothing changes":
  - `(\forall ...; ; ...)`
  - `x.size() == \old(x.size()) && \old(x).containsAll(x)`
  - o ancora meglio se è definita `x.equals(\old(x))`

Design Patterns

Factory / Abstract Factory (strutturale)

- Istanzia una classe astratta, senza riferimenti alle sottoclassi.
- Una `Abstract Factory` è semplicemente un insieme di `Factory`
  (ovvero una `Factory` che produce più di un oggetto)

```
public abstract class Student {
    // not override-able
    public final Exam produceExam()
        { return examFactory(); }
    protected abstract Exam examFactory();
}
public class PreparedStudent extends Student {
    @Override Exam examFactory()
        { return new GoodExam(); }
}
```

Adapter (strutturale)

- Adatta le classi di una libreria ad un'altra

```
public class PDFFile implements printable {
    // LibPDFFile is an external library
    private finale LibPDFFile f;
    @Override void print(int sz)
        { LibPDFFile.stampa(sz * 2); }
}
```

Decorator (strutturale)

- Definisce una classe astratta,
  che contiene un riferimento all'oggetto da decorare

```
public class MsgPrinter {
    public void print(String message)
    { System.out.println(message); }
}
public class Msg2xPrinter extends MsgPrinter {
    private final MsgPrinter printer;
    public Msg2xPrinter(MsgPrinter printer)
    { this.printer = printer; }
    @Override public void print(String message)
    { printer.print(message + " " + message); }
}
```

Command (comportamentale)

- Gestice comandi a priori non conosciuti

```
public interface Game {
    public void jump(); public void run();
}
public class GameController {
    private Game game;
    private final List<GameCtrlBtn> ctrlBtns;
    GameController(int numButtons, Game game) {
        this.game = game;
        this.ctrlBtns = new ArrayList<>(numButtons);
        for (int i=0; i<numButtons; i++)
            { ctrlBtns.add(new GameCtrlBtn()); }
        ctrlBtns.forEach(btn -> btn.setGame(game));
    }
    public void setBtnCmd(int btnId, GameCmd cmd)
        { ctrlBtns.get(btnId).setCommand(cmd); }
    public void changeGame(Game game) {
        this.game = game;
        ctrlBtns.forEach(btn -> btn.setGame(game));
    }
}
public class GameCtrlBtn {
    private GameCmd cmd;
    private Game game;
    public void setGame(Game game) { }
    public GameCmd getCommand() { }
    public void setCommand(GameCmd cmd) { }
    public void onClick()
        { if (cmd != null) { cmd.execute(game); } }
}
public interface GameCmd {
    public void execute(Game game);
}
public class JumpGameCommand implements GameCmd {
    @Override public void execute(Game game)
        { game.jump(); }
}
```

Strategy (comportamentale)

- `Command` dice che azione eseguire,
  mentre `Strategy` come (con quale strategia) eseguirla
  - es: posso sortare in più maniere

```
public abstract class Sorter {
  public void <T extends Comparable<T>>
      sort(List<T>);
}
public class bubbleSorter extends Sorter {
  @Override public void
  <T extends Comparable<T>> sort(List<T>) { }
}
public class myAlgo {
  List<Integer> myList;
  Sorter s;
  public void foo() { s.sort(myList); }
}
```

Observer (comportamentale)

- Un oggetto `Observable` mantiene una lista di `Observer`
- Quando necessario, `Observable` chiama `update()` su ogni `Observer`

```
public interface ClickObsv {
    public void onClick();
}
public class LogClickObserver implements ClickObsv {
    @Override public void onClick() { }
}
public class UserInterface extends ClickSource {
    public void click() { this.notifyClickObsv(); }
}
public abstract class ClickSource {
    private final List<ClickObsv> clickObservers
        = new ArrayList<>();
    public void addClickObsv(ClickObsv observer)
```

```
        { clickObservers.add(observer); }
    public void notifyClickObsv()
        { clickObservers.forEach(ClickObsv::onClick);
    }
}
```

State (comportamentale)

- gestisce azioni da compiere dipendentemente dallo stato

```
public class Phone {
    private PhoneState state;
    public Phone() { }
    public void receiveCall()
        { state.receiveCall(); }
    public void setState(PhoneState state)
        { this.state = state; }
}
public interface PhoneState {
    public void receiveCall();
    public void receiveMessage();
}
public class SilentState implements PhoneState {
    @Override public void receiveCall() { }
    @Override public void receiveMessage() { }
}
```

Model-View-Controller (architetturale)

- `Model`: stato e logica applicativa
- `View`: logia di visualizzazione del modello
- `Controller`: riceve input da `View` e modifice `Model`
  - in alternativa si può usare un `Observer`

Model-View-Presenter

- `View` <–> `Controller` <–> `Model`
- si usano interfacce per far comunicare `View` e `Controller`

```
public class MusicAppView {
    private final List<ClickObsv> obList
        = new ArrayList<>();
    public void addClickObsv(ClickObsv clickObsv)
        { obList.add(clickObsv); }
    public void clickNext() // user click here
        { obList.forEach(ClickObsv::onClick); }
    public void updateCurrentSong(Song s)
        { System.out.println(s.getTitle()); }
}
public interface ClickObsv {
    public void onClick();
}
public class MusicAppCtrl implements ClickObsv {
    private final MusicLibraryModel model;
    private final MusicAppView view;
    public MusicAppCtrl
    (MusicLibraryModel model, MusicAppView view) {
        this.model = model;
        this.view = view;
        view.addClickObsv(this);
    }
    @Override public void onClick() {
        Song newSong = model.getNextSong();
        view.updateCurrentSong(newSong);
    }
}
public class MusicLibraryModel {
    private final List<Song> songs;
    private int nextSong;
    public MusicLibraryModel(List<Song> songs) { }
    public Song getNextSong() { }
}
```

o alterativamente

```
public class MusicAppView
    implements MusicLibraryModelObsv {
    private final List<ClickObsv> obList
        = new ArrayList<>();
    public void addClickObsv(ClickObsv clickObsv)
        { obList.add(clickObsv); }
    public void clickNext()  // user click here
        { obList.forEach(ClickObsv::onClick); }
    private void updateCurrentSong(Song s)
        { System.out.println(s.getTitle()); }
    @Override public void modelChanged(Song song)
        { updateCurrentSong(song); }
}
public interface ClickObsv {
    public void onClick();
}
public class MusicAppCtrl implements ClickObsv {
    private final MusicLibraryModel model;
    public MusicAppCtrl(MusicLibraryModel model,
        MusicAppView view) {
        this.model = model;
        view.addClickObsv(this);
    }
    @Override public void onClick()
    { model.nextSong(); }
}
public class MusicLibraryModel {
    private final List<Song> songs;
    private final List<MusicLibraryModelObsv> obsv;
    private int nextSong;
    public MusicLibraryModel(List<Song> songs,
        MusicAppView view)
        { ...; obsv.add(view); }
    public void nextSong() {
        ...; obsv.forEach(observer
            -> observer.modelChanged(song));
    }
}
```

Model-View-ViewModel

- `ViewModel` (`Observable`) estrae dati da `Model`, ma non contiene riferimenti a `View`
- `View` è l' `Observer`
- I framework, spesso, si occupano di `View` <–> `ViewModel`