Indice

Lambda expression

1.1 Intro

Per comprendere ed usare le lambda, bisogna partire dalle *interfacce*.
Un'interfaccia è un meccanismo per definire un *contratto* tra due parti, il fornitore del servizio (l'interfaccia stessa) e le classi che vogliono che i loro oggetti siano utilizzabili con quel servizio.
Prendiamo ad esempio l'interfaccia Comparable<T>, avente un metodo, compareTo(T o), che restiruisce un intero e assicura di confrontare solo oggetti dello stesso tipo.
Se una classe decidesse di implementare questa interfaccia, quindi fornire un'implementazione del metodo, allora i suoi oggetti potrebbero essere ordinati da java.
Il contratto è che x.compareTo(y) deve restituire:

- un intero positivo se x viene dopo di y;
- un intero negativo nel caso contrario;
- 0 altrimenti.

N.B. La classe String implementa di suo questa interfaccia e implementa compareTo con il confronto lessicografico.
Questo spezzone di codice funziona in quanto *Arrays.sort* riesce a ordinare oggetti la cui classe implementa Comparable e, come detto prima, String la implementa.

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends); // friends is now ["Mary", "Paul", "Peter"]
System.out.println(Arrays.toString(friends));
```

Così come potremmo ordinare oggetti *Employee* in base al loro nome

```
public class Employee implements Comparable<Employee> {
    private String name;

    @Override
    public int compareTo(Employee other) {
        return name.compareTo(other.getName());
    }
}
```

dove, in questo caso, il compareTo di Employee delega il confronto al compareTo di String.
Se volessimo ordinare Employee/String con un altro criterio, non potremmo farlo in quanto non sarebbe possibile definire due metodi compareTo e non sarebbe possibile modificare la classe java.
Esiste una variante di Arrays.sort che oltre ad accettare una lista da ordinare, accetta un'altra interfaccia, Comparator<T>, avente un metodo compare(T o1, T o2) che restituisce un intero.
Quindi per definire un nuovo criterio, dovremo definire una nuova classe, che implementa Comparator, e passarla al metodo Arrays.sort.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new LengthComparator());
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
...
class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

Se il metodo di confronto ci servisse solo in quel punto di SortDemo, saremmo comunque costretti a definire una classe e istanziarla.
Per questo motivo ci sono le *classi anonime*, un meccanismo che riduce la verbosità del codice

- che permette di dichiarare e istanziare una classe allo stesso tempo;

- sono simile alle classi locali, solo che non hanno un nome;
- la loro invocazione avviene come quella di un costruttore, solo che al suo interno c'è una classe vera e propria.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new Comparator<String>() {
            public int compare(String first, String second) {
                return first.length() - second.length();
            }
        });
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
```

Quindi, fino ad ora, abbiamo visto due interfacce, avente un singolo metodo, il contratto che stiamo usando dipende dal quel singolo metodo e se non ci fosse bisogno di mantenersi uno stato per implementare quel metodo, allora sarebbe più comodo poter specificare solo quel singolo blocco di codice invece di creare una classe che implementa l'interfaccia e istanziarla o creare una classe anonima.

1.2 Lambda expression

E' una *funzione anonima*, un blocco di codice che può essere passato, assegnato, restituito in modo da essere eseguito in un secondo momento, una o più volte.
I valori gestiti sono *funzioni* e non oggetti, in java una funzione è un'istanza di un oggetto che implementa una certa interfaccia.
Nell'esempio di LengthComparator, a noi basterebbe dire che, per confrontare due stringhe, bisogna usare il blocco di codice di compare, specificando che first e second sono oggetti di tipo String.
Quindi dovremmo passare ad Arrays.sort una funzione che, dati due oggetti String, restituisce
first.length() − second.length().
In java, la sintassi per definire questa funzione è

```
(String first, String second) -> first.length() - second.length()
```

che risulta essere la nostra lambda expression.
Quindi, nel metodo di Arrays.sort, invece di passare un'istaza di una classe che implementa Comparator o una classe anonima, gli passiamo la lambda.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new Comparator<String>() {
            public int compare(String first, String second) {
                return first.length() - second.length();
            }
        });
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
...
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, (String first, String second) -> first.length() - second.length());
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
```

Il body di una lambda viene eseguito non quando viene passata al metodo sort ma quando bisogna effetivamente confrontare gli oggetto (*esecuzione differita*) e se il body dovesse aver bisogno di più righe, allora si userebbero le parentesi graffe e il return.

Java può inferire il tipo dei parametri della lambda dal contesto, in tal caso si possono omettere i tipi, stessa cosa per il tipo di ritorno anche se qui java fa un controllo che sia utilizzabile nel contesto in cui viene usata la lambda.
Si può assegnare/passare una lambda quando ci si aspetta un oggetto dichiarato di tipo interfaccia

- che ha un singolo metodo astratto;
- purché la lambda sia compatibile con tale metodo, considerando il tipo dei parametri della lambda, che devono essere compatibili coi parametri del metodo, e del tipo inferito del body della lambda che deve essere compatibile col tipo di ritorno del metodo.

Una tale interfaccia è detta *interfaccia funzionale* o *SAM* (Single Abstract Method).

1.3 Method references

Il codice che si scrive in una lambda expression richiama semplicemente un metodo che è già implementato, in questi casi, invece di passare/assegnare una lambda che chiama semplicemente quel metodo passandogli i parametri della lambda, si passa/assegna un riferimento a quel metodo (*method reference*), attraverso la notazione '::'.
Abbiamo tre tipi di method reference

- Class::instanceMethod dove il primo parametro diventa il ricevente del metodo, gli altri sono passati al metodo

```
Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));

Arrays.sort(strings, String::compareToIgnoreCase);
```

- Class::staticMethod dove tutti parametri sono passati al metodo statico

```
list.removeIf(x -> Objects.isNull(x));

list.removeIf(Objects::isNull);
```

- Class:instanceMethod dove il metodo viene richiamato sull'object specificato prima dei :: mentre gli altri parametri sono passati al metodo

```
strings.forEach(x -> System.out.println(x));

strings.forEach(System.out::println);
```

Con i method reference si usa uno stile più dichiarativo, si scrive meno codice e lo si legge meglio.

1.4 Scope di una lambda

Un'interfaccia funzionale può avere tanti metodi statici e di default ma basta che abbiamo un singolo metodo astratto.
Un'interfaccia conviene annotarla con il tag @FunctionalInterface, così facendo il compilatore controllerà che il vincolo sia rispettato e gli altri utenti sapranno che quell'interfaccia è pensata come funzionale.
Non possiamo dichiarare variabili locali in una lambda o parametri di una lambda con lo stesso nome di variabili già definite nei blocchi esterni.
Una lambda può riferirsi a variabili definite NON dentro la lambda purchè dichiarate final o effectively final, si dice che ha un *ambito di visibilità circostante* (enclosing scope), per esempio

```
String message = "Hello ";
repeat(10, (x) -> System.out.println(message + x));
```

La lambda si riferisce alla variabile 'message' ma il body della lambda sarà effettivamente eseguito da dentro il metodo repeat e, da dentro il metodo repeat, la variabile 'message' non è visibile, eppure il codice è lecito e tutto funziona.
Una lambda ha tre ingredienti, parametri, body e i valori delle *variabili libere*, ovvero parametri che non fanno parte dei parametri della lambda e che non fanno parte delle variabili dichiarate nel blocco di codice della lambda.
Nell'esempio di prima, 'x' non è una variabile libera, è legata al parametro della lambda, mentre 'message' si.
Una lambda expression cattura il valore di queste variabili libere e, quando viene eseguita, il body è chiuso rispetto ad esse ed è per questo motivo che a runtime una lambda è detta *chiusura* (closure).
Ovvero prima di passare la lambda a repeat, è come se java sostituisse message direttamente con "Hello ".

Stream

Una “vista” dei dati per specificare computazioni a un più alto livello concettuale rispetto alle collezioni e gli iteratori.
Supponiamo di voler calcolare la media dei salari di una collezione di oggetti Employee.

Se decidessimo di usare gli iteratori

- dovremmo dichiarare una variabile per accumulare i risultati intermedi;
- iterare sulla sorgente dati, dove, ad ogni iterazione, aggiorniamo i risultati intermedi;
- alla fine, calcolare la media.

Se usassimo gli stream, basterebbe

- specificare la sorgente dati;
- la proprietà di interesse;
- cosa vogliamo fare con quella proprietà.

La libreria stream farà tutto il resto ottimizzando il calcolo.

Rispettano il principio ”“what, not how”, ovvero si specifica cosa si vuole fare e non il come deve essere fatto.
Stream e collezioni, superficialmente, sembrano simili, entrambi permettono di trasformare e recuperare dati ma

- uno stream non *memorizza* i dati, sono memorizzati nella collezione originale o generati su richiesta;
- uno stream non *modifica* i dati originali, ma genera un nuovo stream dove alcuni elementi dello stream precendete non sono presenti nello stream corrente;
- le operazioni sono ”*lazy*” quanto il più possibile, ovvero non vengono eseguite finché non serve il risultato (si possono avere anche stream infiniti).

Gli stream si basano sul *method chaining*, ovvero chiamano un metodo sul risultato di un altro metodo, senza memorizzare i risultati intermedi.
Il workflow tipico di uno stream consiste nel creare una “pipeline” di operazioni in 3 fasi:

- creazione dello stream;
- specifica delle operazioni intermedie;
- applicazione di un’operazione finale di riduzione per produrre un risultato oppure un’operazione di raccolta.

2.1 Creazione di uno stream

Gli stream si creano da collezioni e array o usando generatori o iteratori.
Per le collezioni abbiamo i metodi stream() e parallelStream(), per gli array o un numero arbitrario di argomenti (vararg) abbiamo il metodo statico Stream.of() e poi abbiamo i metodi Stream.generate(Supplier <T>) e Stream.iterate(T seed, UnaryOperator<T> f).
Gli ultimi due metodi possono genera stream infiniti.

2.1.1 Stream infiniti

Stream.generate(Supplier <T>) prende una lambda senza argomenti e viene chiamato solo quando viene richiesto allo stream il prossimo elemento

```
Stream<String> echos = Stream.generate(() -> "Echo");
Stream<Double> randoms = Stream.generate(Math::random);
```

Stream.iterate(T seed, UnaryOperator<T> f) dove seed è il ”seme” iniziale mentre f è una funzione che sarà applicata al valore precendete

```
Stream<BigInteger> integers = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE))
```

2.2 Operazioni intermedie

Le operazioni intermedie di uno stream sono metodi che trasformano lo stream in un altro stream.
Tra questi metodi troviamo filter() che si occupa di filtrare uno stream, map() che si occupa di trasformare gli elementi di uno stream, limit() che limita ad un certo numero di elementi dello stream (occhio a dove lo mettiamo), skip() che salta determinati elementi dello stream, distinct() che scarta i duplicati, etc...
Con gli stream si scrive meno codice, lo si rende più leggibile ma allo stesso tempo è molto facile commettere errori, i due prossimi stream forniscono un risultato differente

```
#SI LIMITA ALLE PRIME 5 STRINGHE CHE RISPETTANO IL FILTRO
long count = words.stream()
                  .filter(w -> w.length() > 12)
                  .limit(5)
                  .count();
#SI LIMITA ALLE PRIME 5 STRINGHE
long count = words.stream()
                  .limit(5)
                  .filter(w -> w.length() > 12)
                  .count();
```

2.3 Operazione di riduzione di uno stream

Una volta applicate tutte le operazioni di trasformazione, alla fine si deve applicare un'operazione di riduzione.
Alcuni metodi di riduzione sono min() e max() che restituiscono, rispettivamente, il minimo ed il massimo di uno stream in base ad un criterio oppure il metodo reduce(...) che prende una funziona binaria e continua ad applicarla partendo dai primi due elementi.

```
List<Integer> values = ...;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

Le operazioni di raccolta/riduzione forzeranno l'esecuzione di tutte le operazioni, lazy, precedenti e dopo lo stream non potrà più essere usato.
Alcuni di queste operazioni, come reduce(...), restituiscono un oggetto di classe Optional, un'alternativa più sicura della gestione dei valori null come ad esempio findFirst(), ifPresent(), etc...

2.3.1 Optional

Un oggetto Optional<T> è un wrapper di un oggetto T oppure nessun oggetto.
L'idea di base sull'utilizzo di un oggetto Optional è quella di usare i suoi metodi che permettono di produrre un'alternativa se il valore non è presente o consumarlo.
Questo oggetto deve essere usato in modo appropriato, altrimenti si hanno gli stessi problemi che si hanno con null, ad esempio

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod();

T value = ...;
value.someMethod();
```

Questi due blocchi sono uguali, se optionalValue non contenesse alcun valore, allora avremo una NPE, stessa cosa per quest'altro esempio

```
if (optionalValue.isPresent())
    optionalValue.get().someMethod();

if (value != null)
    value.someMethod();
```

2.4 Operazione di raccolta di uno stream

Un'alternativa alla riduzione di uno stream è la raccolta.

Ci sono molti metodi tra cui il metodo collect() che prende in input oggetti che implementano l'interfaccia Collector, tra cui Collectors che fornisce metodi per creare istanze delle collezioni più comuni come ad esempio toList() o toSet(), iterator() che ritorna un iteratore o anche toArray().
Spesso si vuole una mappa che associa ad una chiave, una collezione di oggetti e qui entrano in gioco i metodi di raggruppamento o partizionamento.
Si usa il primo specificando la lambda che calcola la chiave, detta *classifier function* mentre si usa il secondo quando quest'ultima è booleana

```
#per ogni stringa del nome abbiamo la lista delle relative persone
Map<String, List<Person>> sameName = people.collect(Collectors.groupingBy(Person::getName));

#abbiamo due liste, una che contiene il nome e una che non lo contiene
Map<Boolean, List<Person>> emptyNames = people.collect(Collectors.partitioningBy(p ->
    p.getName().isEmpty()));
```

Di default groupingBy raggruppa in una List ma ci sono altre versioni che permettono di specificare un downstream collector

```
#visto prima
Map<String, List<Person>> sameName = people.collect(Collectors.groupingBy(Person::getName));

#nuovo tipo
Map<String, Long> sameName = people.collect(Collectors.groupingBy(Person::getName, Collectors.counting()));
```

INFORMATION HIDING

Utilizzo di meccanismi per evitare che tutti possano accedere a certe parti di codice.
Noi abbiamo visto i livelli di accessibilità in Java

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Per convenzione

- le variabili d'instanza dovrebbero essere private (evita anche protected);
- i campi static, per rappresentare costanti accessibili a tutti, dovrebbero essere public final;
- i metodi che utilizzano dettagli interni dovrebbero essere private;
- i metodi che potrebbero servire alle sottoclassi, dovrebbero essere protected;
- i metodi che saranno usati dalle altre classi, dovranno essere public.

Creare un oggetto che ha parametri d'instanza, privati, che però vi si possonono accedere, con un get, o settare, tramite un setter, è la cosa sbagliata.
Supponiamo di avere una classe Person tale che

```
public class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  @Override
  public String toString() {
    return "Person [name=" + name + "]";
  }
}
```

Se instaziassi la classe e cercassi di accedere al campo name, il compilatore mi segnalerebbe errore.
L'information hiding è pensato per essere usato a tempo di compilazione e non a run time.
Java permette di accedere alle variabili, anche private, di una classe.
infatti

```
@Test public void testReadPrivateMember() throws ... {
  Person person = new Person("John");
  // riferimento al campo (anche se privato)
  Field nameField = person.getClass().getDeclaredField("name");
  // aggira il controllo di accesso
  nameField.setAccessible(true);
  // legge il valore del campo
  Object nameValue = nameField.get(person);
  assertEquals("John", nameValue);
}
```

N.B. Il metodo setAccessible(boolean bol) permette, effettivamente di accedere ai campi private, altrimenti non sarebbe possibile farlo.
Se io dichiarassi name public, legittimerei il suo uso da parte dei cliente, imponendomi il vincoli non poter mai più modificare quella variabile, altrimenti i client verranno rotti (non compileranno più).

Ogni client che userà quel membro sarà strettamente accoppiato (tightly coupled) a quel membro (noi questo non lo vogliamo).
Stessa cosa per protected, solo che qui saranno le sottoclassi ad essere tightly coupled.
Dichiarando un membro private, lo nascondo al "mondo", non legittimiamo il suo uso da parte dei client.
Il compilatore controllerà il corretto uso nei vari client e

- se compileranno con errori di accesso, significa che i client non sono legittimati all'uso;
- se accederanno al membro via reflection (run time) e qualcosa non dovesse funziona, sarà colpa loro.

Più dettagli interni nascondiamo, più avremo disaccoppiamento e più sarà facile testare singole componenti in isolamento.
Se diachiarassimo qualcosa come public/protected, come le API, e volessimo effettuare dei cambiamento, dovremmo notificare i client quando rilasceremo il nuovo aggiornamento, specificando se si tratta di qualcosa che rompe API, implementa qualcosa di nuovo oppure la risoluzione di bug.
Un modo per indicare questi cambiamenti è l'utilizzo della semantic versioning, un numero composta da tre parti, $X.Y.Z$, dove se

- dovesse cambiare X, significherebbe aver introdotto qualcosa che rome l'API;
- dovesse cambiare Y, significherebbe aver introdotto qualcosa di nuovo;
- dovesse cambiare Z, significherebbe aver risolto dei bug (sperando di non aver introdotto nuovi bug).

3.1 Design Pattern VS Design Principle

Un design pattern sbalisce la linea guida su cosa è giusto e su cosa è sbagliato quando si fa il design di un'applicazione, dice cosa fare (e non fare) e non come farlo.
Un design pattern è una soluzione generica e riusabile per un problema comune, dicono come risolvere un problema in un certo contesto software, fornendo chiare linee guida.

ACRONIMO SOLID

I principi SOLID prescrivono come organizzare funzioni e dati in classi e come tali classi dovrebbero essere interconnesse.
Lo scopo di questi principi è la creazione di strutture software che tollerano il cambiamento, sono facili da comprendere/testare e sono riusabili in diversi sistemi software.

4.1 Single Responsability Principle (SRP)

Prevede che una classe deve avere una sola responsabilità, ovvero deve avere una e una sola ragione per essere modificata.
Si dice che la classe deve essere

- *coesiva*, ovvero quanto le cose di un certo gruppo hanno ragione di stare insieme;
- *loosely coupling*, ovvero deve dipendere il meno possibile dai metodi di altre classi.

Se una classe è responsabile di più cose, la si dovrà cambiare più spesso e cambiare frequentemente una classe porta ad avere ripercussioni su tutte quelle classi che dipendono da lei.
Ciò comporterà, ad ogni modifica, nuova ricompliazione e nuovo test dei metodi.
Prima di aggiungere qualcosa di nuovo a una classe ci si dovrebbe interrogare su quale sia la responsabilità di questa classe, se la risposta comprende funzionalità scorrelate congiunte con un "e" o un "oppure" allora, probabilmente, si sta violando l'SRP.

4.2 Open-Closed Principle (OCP)

Il principio aperto/chiuso stabilisce che le classi debbano essere aperte alle estensioni e chiuse alle modifiche.
Per modifica si intende il cambiamento del codice di una classe esistente ed estensione significa aggiungere nuove funzionalità.
Tutto ciò sta a significare che dovremmo essere in grado di aggiungere nuove funzionalità senza toccare il codice attuale della classe, questo perché ogni volta che modifichiamo il codice, rischiamo di dare vita a potenziali bug.
Se nell'SRP si cerca di decomporre la responsabilità, qui si cerca di capire quali parti devono essere concrete (da implementare) e quali devono essere astratte (saranno implementate dai consumatori del software).
Negli OOP abbiamo a disposizione il meccanismo dell'ereditarità per creare una classe derivata da una già esistente, il meccanismo dell'override per ridefinire un metodo della classe padre e la funzionalità del binding dinamico che permette, dato un oggetto, di chiamare il metodo giusto a runtime.
Questo, però, non è detto che basti a rispettare OCP, specialmente se si estende una classe concreta o se la classe derivata si basa, fortemente, su dettagli implementatitivi della superclasse. Per esempio, voglio estendere una classe, MySet, per contare gli elementi inseriti

```
public class MySet<E> {
  public void add(E o) {
  // add the element to the internal set
  }

  public void addAll(Collection<E> c) {
    for (E e : c) {
      add(e);
    }
  }
}
```

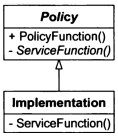
```
public class MyCountingSet<E> extends MySet<E> {
  private int count = 0;

  @Override
  public void add(E o) {
    super.add(o);
    count++;
  }
}
```

Se io modificassi MySet, ad esempio modifico addAll(), non chiamando più dentro add(), allora la mia classe derivata non sarebbe più corretta, in quanto non avremo più un aumento del contatore.
Quindi, per risolvere questo problema, devo modificare la mia classe, facendo override anche del secondo metodo

```
public class MySet<E> {
  public void add(E o) {
  // add the element to the internal set
  }

  public void addAll(Collection<E> c) {
    // add directly all elements
    // WITHOUT relying on add
  }
}
```

```
public class MyCountingSet<E> extends MySet<E> {
  private int count = 0;

  @Override
  public void add(E o) {
    super.add(o);
    count++;
  }

  @Override
  public void addAll(Collection<E> c) {
    super.addAll(c);
    count += c.size();
  }
}
```

Inoltre, se ad un certo punto tornassi alla versione originale di MySet, avrei lo stesso problema, ovvero MyCountingSet non funzionerebbe più in quanto il contatore verrebbe incrementato due volte ogni volta.
Quindi possiamo vedere che le modifiche fatte a MySet possono anche essere giuste o minimali ma sono devastanti per la mia sottoclasse.
Questo porta al concetto della **fragile base class problem**, ovvero una classe è considerata fragile se piccole modifiche corrette e sicure alla classe base, possono portare le classi derivate a non funzionare più correttamente.
Quindi sarebbe meglio proibire l'estensione della classe (con un final), proibire la ridefinizione di alcuni metodi (anche qui final) e documentare come una classe base dovrebbe essere usata/estesa.
Allora, dal concetto di classi concrete ed ereditarietà, arriviamo al concetto di classi astratte o interfacce ed, eventualmente, al principio di composizione e delega.
Prediamo come esempio la classe astratta Policy che ha un metodo, public e probabilmente final, PolicyFunction che, a sua volta, implementa una certa politica in termini di un metodo, protetto e astratto, ServiceFunction.
Policy, quindi, ha una parte concreta (PolicyFunction) ed una astratta (ServiceFunction), si dice che è aperta all'estensione, tramite la parte astratta e chiuse alle modifiche dovute al fatto che PolicyFunction è final.



Supponiamo di avere una classe Client che si occupa di fare logging e di voler aggiungere un nuovo tipo di log, basterà, quindi, aggiungere un nuovo tipo di logging ad enum ed un nuovo ramo allo if-else.
Come si vede, la classe è si configurabile rispetto al log ma non è aperta ad estensioni del log in quanto la sua gestione dipende da un if-else/switch e quanfo si usa un if-else/switch, si sta violando OCP, inoltre c'è il dubbio su cosa faccia l'ultimo else (in questo caso non fa nulla).

```
public class Client {
  public enum LogConf { CONSOLE, FILE }
  private LogConf logConf;

  public Client(LogConf logConf) { this.logConf = logConf; }

  public void aMethod() {
    // do something before or after the log method...
    log("a message");
  }

  private void log(String message) {
    if (logConf == LogConf.CONSOLE) {
    // log to System.out
    } else if (logConf == LogConf.FILE) {
    // log to a File
    } else {
    // what to do?!?
    }
  }
}
```

Quindi, per risolvere ciò

- decidiamo di astrarre la funzionalità di logging attraverso l'interafccia Log che avrà diverse implementazioni come ConsoleLog o FileLog;
- la classe Client dichiara una variabile privata di tipo Log;
- l'istanza Log viene passata al costruttore;
- Client delega completamente il logging all'istanza di Log.

La nuova classe è estendibile a nuovi sistemi di logging, infatti basta creare una nuova implementazione di Log, il Client non è stato minimamente toccato, non c'è stato bisogno di ricompilare.

```
public class Client {
  private Log logger;

  public Client(Log log) {
    this.logger = log;
  }

  public void aMethod() {
    // do something...
    log("a message");
    // do something...
  }

  private void log(String message) {
    logger.log(message);
  }
}
```

```
public interface Log {
  public void log(String message)
}

public class ConsoleLog implements Log {...}

public class FileLog implements Log {...}
```

Ovviamente le due classi concrete fanno override del metodo log.
Esempio pratico sono i software basati sui plug-in, ovveri software che non è possibile modificare ma estendibili tramite plug-in.

4.3 Liskow Substitution Principle (LSP)

Questo principio si basa sul concetto di sottotipo.
In java, un oggetto di tipo T sottotitpo di S, detto supertipo, può essere sempre

- assegnato a una variabile dichiarata di tipo S;
- passato come argomento di un metodo che si aspetta un parametro di tipo S;
- restituito in un metodo il cui tipo di ritorno è S.

Per un sottotitpo valgono la proprietà riflessiva, ovvero data una classe A, A è sottotipo di se stessa (si indica con $A <: A$) e la proprietà transitiva, ovvero siano A, B e C tre classi, se $A <: B$ e $B <: C \Rightarrow A <: C$.
Il prinicipio di Liskow è qualcosa di più forte della definizione di sottotipo, ovvero, vuole che il comportamento del programma, quando sostituisco S con T, non cambia, non si rompe, continua a funzionare correttamente (non significa che non cambia nulla, altrimenti non avrebbe senso).
Per esempio, ho una classe Rectangle con due campi, height e weight, metodi setter ed un metodo area.

```
public class Rectangle {
  private int height, width;

  public void setHeight(int height) {...}

  public void setWidth(int width) {...}

  public int area() { return height * width;}
}
```

```
@Test
public void testRectangle() {
  Rectangle r = new Rectangle();
  r.setHeight(5);
  r.setWidth(3);
  assertEquals(15, r.area());
}
```

Ora voglio creare la classe Square da rettangolo, tanto basta porre entrambi i lati uguali.

```
public class Square extends Rectangle {

@Override
  public void setHeight(int height) {
    super.setHeight(height);
    super.setWidth(height);
  }

  @Override
  public void setWidth(int width) {
    super.setWidth(width);
    super.setHeight(width);
  }
}
```

```
@Test
public void testSquare() {
  Square s = new Square();
  s.setHeight(5);
  assertEquals(25, s.area());
  s.setWidth(3);
  assertEquals(9, s.area());
}
```

Presi singolarmente, i test passano, ora, però, bisogna vedere che, se applicando il principio di sostituzione, tutto rimane invariato.

```
@Test
public void testSquareAsRectangle() {
  Rectangle r = new Square();
  r.setHeight(5);
  r.setWidth(3);
  assertEquals(15, r.area());
}
```

Questo test fallisce perchè, quando chiamo in sequenza i due metodi setter, questi saranno chiamati da quadrato e non da rettangolo (binding dinamico).
Quindi Square non è sostituibile a Rectangle anche perchè alla classe Square, alla fine dei conti serve un solo campo.
Per risolvere il problema possiamo pensare di eliminare i due metodi setter ed introdurre un'interafccia, Shape, con all'interno la definizione del metodo area.
Così facendo, entrambe le classi concrete, implementeranno Shape e ridefineranno, a modo loro, il metodo area attraverso override.

```
public interface Shape {
  public int area();
}
```

```
public class Rectangle implements Shape {
  private int height, width;

  public Rectangle(int height, int width) {
    this.height = height;
    this.width = width;
  }

  @Override
  public int area() { return height * width;}
}
```

```
public class Square implements Shape {
  private int side;

  public Square(int side) {
    this.side = side;
  }

  @Override
  public int area() { return side * side;}
}
```

I test che useranno Shap, passandogli prima Rectangle e poi Square, funzioneranno

```
@Test
public void testShape() {
  Shape s = new Rectangle(5,3);
  assertEquals(15, s.area());
  s = new Square(5);
  assertEquals(25, s.area());
}
```

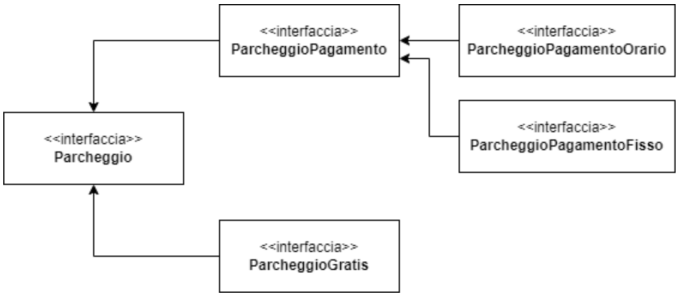
Detto in parole povere, bisogna cercare di lavorare, il più possibile, con classi astratte o interfacce, lavorare verso l'astrazione e non verso l'implementazione.

4.4 Iterafce Segregation Principle (ISP)

È simile all'SRP ma incentrato su interfacce e client.
Supponiamo di avere a disposizione un'interfaccia contente metodi che svolgono differenti compiti.
Un client che implementerà questa interfaccia, sarà costretto ad implementare tutti i suoi metodi, anche quelli che non gli servono.
Se un client richiedesse una modifica all'interfaccia, anche gli altri client ne sarebbero influenzati, anche se la modifica dovesse riguardare un metodo che a loro non serve.
Quindi sarebbe conveniente dividere l'interfaccia originale in interfacce più piccole, così facendo l'interfaccia originale conterrà solo i metodi comuni ai client, mentre le interfacce più piccole si occuperanno dei metodi specifici.
Prendiamo ad esempio il sistema di pagamento di un parcheggio

```
public interface Parcheggio {

  void parcheggiaAuto(); // Diminuisce parcheggi vuoti di 1
  void esceAuto(); // Aumenta parcheggi vuoti di 1
  void getCapienza(); // Restituisce capienza auto
  double calcolaQuota(Auto auto); // Restituisce il prezzo in base al numero di ore
  void faiPagamento(Auto auto);
}

class Auto {
  ...
}
```

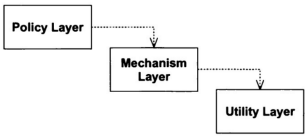
Immaginiamo di voler implementare un parcheggio gratuito (che estenderà Parcheggio) e notiamo che abbiamo obbligato questa classe ad implementare i metodi di pagamento del parcheggio anche se non ne avrebbe bisogno (è gratis) e magari, nel metodo pagamento, lanciamo un'eccezione gestita con un semplice "il parchegio è gratis".
Quindi si nota che l'interfaccia Parcheggio si occupa di due logiche distinte, quella del parcheggio e quella del pagamento del parcheggio stesso.
Una possibile soluzione, quindi, sarebbe quella di scindere l'interfaccia parcheggio in altre due interfacce, quella a pagamento e quella gratis, la prima oltre ad ereditare i metodi riferiti al parcheggio, aggiungerà i metodi riguardanti i pagamenti, mentre la seconda, implementrà l'interfaccia padre.
Con questo nuovo modello, possiamo anche andare oltre e dividere l'interfaccia pagamento per supportare diverse modalità di pagamento.



4.5 Dependency Inversion Principle (DIP)

Questo principio ci dice che i sistemi più flessibili sono quelli dove le dipendenze del codice sorgente si riferiscono solo alle astrazioni.
Per dipendenza nel codice sorgente si intende che, dato un file Client.java, se in dato file nomino un tipo A, allora diremo che il codice sorgente di Client dipende da A.
Anche se in un dato Client.java useremo un sottotipo di A, B, diremo che Client dipende, staticamente, da A.
Da qui utilizzeremo il concetto di modulo/componente inteso come raggruppamento di classi.
Tradizionalemente, i moduli di alto livello dipendono dai moduli di basso livello, per dipendere si intende chiamare direttamente codice di basso livello o istanziare direttamente classi concrete.
Per codice di alto livello, detto anche core, intendiamo la parte di un'applicazione computazionale, algoritmica o che elabora, ovvero è la parte che contraddistingue un'applicazione.
Mentre per codice di basso livello intendiamo interfaccia utente o database.
Quindi se il codice di alto livello dipende dal codice di basso livello, significa che modifiche a quest'ulitmo, avranno un impatto sul codice di alto livello.

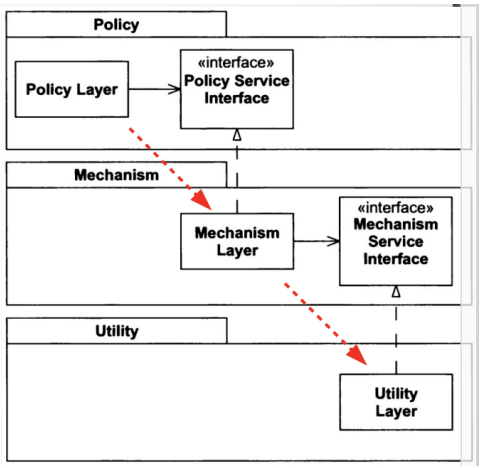
Si prende come esempio un'architettura OO dove la parte più alta indica codice di alto livello.



Il layer di alto livello usa solo il layer sottostante ma usare significa che dipende dal layer sottostante.
Inolte se il layer sottostante dipende dal layer di basso livello, allora il layer di alto livello, transitivamente, dipende dal layer di basso livello.
Per risolvere questo problema

- ogni layer di alto livello dovrebbe dichiarare una propria interfaccia per i servizi di cui ha bisogno e farla implementare al layer sottostante;
- il layer sottostante implementerà l'interfaccia dichiarata dal layer superiore.

Così facendo, le dipendenze vengono invertite, ovvero sono i layer di basso livello a dipendere dai layer di alto livello.



staticamente Policy chiama un metodo dell'interfaccia, a runtime verrà invocato l'implementazione di tale metodo dal layer inferiore, così facendo, posso cambiare un layer sottostante senza modificare un layer soprastante e posso testare un layer soprastante senza avere un layer sottostante ed inoltre è buona cosa non menzionare mai il nome di qualcosa che è concreto (non astratto) e volatile (che cambia spesso) e non avervi riferimento.
Quindi, i design pattern guidano lo sviluppo del codice e il refactoring aiutando a sviluppare software pulito e facile da estendere, riusare, mantenere e testare.

COVARIANZA E CONTROVARIANZA

Dalla matematica, una funzione *f*, tale che f: $D \to C$, può essere sosituita da una funzione *g*, tale che g: $D' \to C'$, sse $C' <: C$ (**covariante** sul tipo di **ritorno**) e $D <: D'$ (**controvariante** su tipo del **parametro**), ovvero una funzione g, che deve sostituire f, può essere più specifica sul tipo di ritorno e più generale sul parametro in input.
Supponiamo di avere due classi, Employee e Manager, tale che Manager<:Employee.
Employee ha un metodo setSalary() e Manager ha il metodo setBonus(int).
Prendiamo altre due classi, EmployyeDB e ManagerDB, tale che ManagerDB<:EmployyeDB.
La prima ha al suo interno un metodo find(int) che restiusce un Employee, mentre il secondo ridefinisce il metodo facendo tornare un Manager.

```
public class EmployeeDatabase {
    public Employee findEmployee(int id) {
        ...
    }
}
```

```
public class ManagerDatabase extends
    EmployeeDatabase {
    @Override
    public Manager findEmployee(int id) {
        ...
    }
}
```

Se eseguissi questo codice, tutto funzionerebba.

```
EmployeeDB d = new ManagerDB();
Employee e = d.find(1);
```

Per il binding dinamico, d staticamente è EmployeeDB ma a runtime è ManagerDB, inoltre il metodo find resituisce un Manager e non abbiamo problemi anche perchè è un sottotipo di Employee.
Questo è consistente col tipo di ritorno che è covariante e che può essere uguale o più specializzato.
Ipotizziamo che il concetto di covarianza valga anche per i parametri e quindi supponiamo che EmployeeDB abbia un metodo updateEmployee(Employee), che setta il salario di un Employee, e che la classe sottotipo, ridefinisca il metodo cambiandone il parametro in un Manager, e aggiungendo al suo interno il metodo setBonus(int) (può farlo in quanto è di Manager).

```
public class EmployeeDB {
...
    public void updateEmployee(Employee e) {
        e.setSalary(...);
    }
}
```

```
public class ManagerDB extends EmployeeDB {
...
    @Override
    public void updateEmployee(Manager e) {
        super.updateEmployee(e);
        e.setBonus(0);
    }
}
```

Eseguendo questo codice

```
EmployeeDatabase d = new ManagerDatabase();
d.updateEmployee(new Employee());
```

Come detto prima, per il binding dinamico, d staticamente è EmployeeDB ma a runtime è ManagerDB, quindi il metodo che sarà chiamato, sarà quello di ManagerDB che però non esiste e quindi lancerà un'eccezione del tipo NoSuchMethodException.
A livello di compilazione, tutto è ok perchè il metodo updateEmployee(Employee), staticamente è di tipo EmployeeDB ma a runtime chiamerà il metodo di ManagerDB dove abbiamo deciso di mettere in posizione covariante anche i parametri e questo non va bene perchè, per il binding dinamico, verrebbe chiamato updateEmployee di ManagerDatabase che peraltro si aspettava un Manager ed invece si ritrova un Employee e boom, eccezione del tipo NoSuchMethodException.
Java adotta un sistema di tipi **sound**, ovvero un programma, accettato staticamente dal compilatore, non generarà mai errori NoSuchMethodException a runtime.
Nello specifico i tipi statici, controllati durante la compilazione, durante l'esecuzione si mantengono come tali oppure diventano sottotipi.
Java non permette la controvarianza durante la ridefinizione di un metodo in quanto richiede che i tipi dei parametri non cambino affatto (si dice che java è invariante sul tipo dei parametri).

Supponiamo che Employee abbia un supertipo, Person e che EmployeeDB sia esteso da un'altra classe, PersonDB che esegue override del metodo chiamando dentro il suo metodo setName().

```
public class EmployeeDatabase {
    ...
    public void updateEmployee(Employee e) {
        e.setSalary(...);
    }
}

public class PersonDataBase extends EmployeeDatabase {
    @Override
    public void updateEmployee(Person e) {
        super.updateEmployee(e);
        e.setName("");
    }
}
```

Dal punto di vista della controvarianza, ciò è corretto, però avremo un errore, in quanto non potremmo usare il metodo super perchè richiede Employee, mentre gli stiamo passando un Person e un Person NON è un Employee. Quindi possiamo dire che Java, per la ridefinizione di metodi, adotta l'approcio sicuro, ovvero covariante sul tipo di ritorno e invariante sui parametri.
Magari una soluzione potrebbe essere quella di preferire l'overloading all'override, magari abbiamo lo stesso metodo, uno che prende in input Employee e l'altro che prende Person.

```
public class EmployeeDatabase {
    ...
    public void updateEmployee(Employee e) {
        e.setSalary(...);
    }
}

public class ManagerDatabase extends EmployeeDatabase {
    ...
    public void updateEmployee(Manager e) {
        super.updateEmployee(e);
        e.setBonus(0);
    }
}
```

Anche qui ci saranno problemi, infatti dato il seguente spezzone di codice

```
EmployeeDatabase d1 = new ManagerDatabase();
ManagerDatabase d2 = new ManagerDatabase();

// chiama EmployeeDatabase.updateEmployee(Employee)
d1.updateEmployee(new Employee());

// chiama ManagerDatabase.updateEmployee(Manager)
d2.updateEmployee(new Manager());

/*  chiama ANCORA EmployeeDatabase.updateEmployee(Employee) perche' d1 e' staticamente EmployeeDatabase
 *   ManagerDatabase.updateEmployee(Manager) aggiunge un metodo in overloading ma NON ridefinisce
 *   EmployeeDatabase.updateEmployee(Employee) quindi non si applica il binding dinamico!
*/
d1.updateEmployee(new Manager());
```

Quindi potremmo risolvere passando dall'usare l'overloading in una gerarchia di classi, all'usare l'overloading in una singola classe.

```
public class EmployeeDatabase {
    ...
    public void updateEmployee(Employee e) {
        e.setSalary(0);
    }

    public void updateEmployee(Manager e) {
        e.setBonus(0);
    }
```

```
}
```

però anche nel prossimo spezzone, avremo un comportamento non prevedibile

```
EmployeeDatabase d1 = new ManagerDatabase();

// chiama updateEmployee(Employee): l'argomento e' un Employee
d1.updateEmployee(new Employee());

// chiama updateEmployee(Manager): l'argomento e' un Manager
d1.updateEmployee(new Manager());

Employee e1 = new Employee();
Employee e2 = new Manager();

// chiama updateEmployee(Employee)
d1.updateEmployee(e1);

// chiama sempre updateEmployee(Employee)
d1.updateEmployee(e2);
```

perchè in java, l'overloading è un meccanismo statico.
Implementare una sorta di overloading dinamico è difficile dal punto di vista dell'efficienza (diverso dalla complessità costante del binding dinamico).
Quindi java, per la ridefinizione dei metodi, adotta l'approccio sicuro, covariante sul tipo di ritorno e invariante sui tipi dei parametri.
Per le lambda invece si, possiamo essere controvarianti sui parametri, esempio il metodo map degli stream che è definito come

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

ovvero, una lambda di tipo Function<T, R> può accettare anche una lambda di (sotto)tipo Function<T', R'> dove T <: T' e R' <: R

DESIGN PATTERN

Sono una soluzione generica e riusabile per un problema comune, ci dicono come risolvere un problema in un certo contesto software, fornendo chiare linee guida.
I pattern sono schemi, modelli che descrivono relazioni tra interfacce e classi e interazioni tra oggetti.
Sono pensati per risolvere specifici problemi di design, rendendolo flessibile, elegante e riusabile.
Ogni pattern è composto da 4 elementi, il *nome* per riferirsi al pattern, il *problema* che ci dice quando si applica e non (in alcuni casi ci sono delle condizioni da soddisfare), la *soluzione* che descrive gli elementi (classi, interfacce, oggetti) e le loro relazioni (responsabilità e collaborazioni) e le *conseguenze* che mostrano i risultati e i compromessi dall'applicazione del pattern.
Alcuni pattern sopperiscono alle mancanze di funzionalità/caratteristiche del linguaggio di programmazione (ad esempio il Visitor sopperisce alla mancanza dell'overloading dinamico).
Si differenziano per **purpose** (proposito) che riguardano le tipologie di pattern (creazionali, strutturali e comportamentali) e per **scope** (campo di azione) che riguardano le relazioni tra classi e sottoclassi (ereditarietà e overriding) e oggetti (object composition e delegation).
I class pattern trattano relazioni fra classi e sottoclassi, tali relazioni sono stabilite tramite inheritance (relazioni statiche e fisse a tempo di compilazione), mentre gli object pattern trattano relazioni fra oggetti che possono essere modificate a run-time (relazioni dinamiche).
Nei pattern creazionali i class patterns delegano a sottoclassi mentre gli object patterns delegano a un altro oggetto.
Nei pattern strutturali i class patterns usano l'inheritance per comporre classi mentre gli object patterns descrivono modi per assemblare oggetti.
Nei pattern comportamentali i class patterns usano l'inheritance per descrivere algoritmi e il "flow of control" mentre gli object patterns descrivono come un gruppo di oggetti cooperano per eseguire un certo task.

6.1 Notazione Uml

Unified Modeling Language, una notazione formale per la specifica, costruzione, visualizzazione e documentazione del modello di un sistema software.
Utile sia per documentazione che nella fase di sviluppo.

6.2 Interaction diagram

Mostra l'ordine in cui le richieste fra oggetti vengono eseguite.