PROGRAMMAZIONE CONCORRENTE E DISTRIBUITA

2018/2019

creato dal leggendario Roberto

Solitamente è bene commentare adeguatamente il codice che si scrive e java ha dei comandi speciali che, se usati all'interno di commenti, possono generare documentazione per tale metodo.

```
/**
  * Returns the element at the specified position in this list
  *
  * <p>This method is <i>not</i> guaranteed to run in constant time.
   In some  implementations it may run in time proportional to the
   element position.
  *
  * @param index index of element to return; must be non-negative
  *        and less than the size of this list
  * @return the element at the specified position in this list
  * @throws IndexOutOfBoundsException if the index is out of the
   range
  *         ({@code index < 0 || index >= this.size()})
  */
 E get(int index)
```

Da notare che nel commento viene utilizzato l'HTML per mettere il corsivo qualche parte di testo. Ci sono vari tag che possono tornare utili quando si genera la documentazione via javadoc, ovvero:

- `@param`: utilizzato per descrivere i parametri in input della funzione
- `@return`: utilizzato per descrivere cosa ritorna il metodo in questione
- `@throw`: se il metodo lancia qualche tipo di eccezione che va catturata

In Java tutte le classi derivano da una classe base chiamata Object; quindi quando ad esempio definisco una classe che si chiama semplicemente `public class Test {}` sto in realtà creando una sottoclasse di Object. Il codice di esempio sopra è equivalente a `public class Test extends Object{}`.

Importante notare la differenza che c'è fra il metodo `equals(Object o)` e l'operatore ==.

- L'operatore == confronta le *references*, ovvero guarda se due variabili puntano alla stessa locazione di memoria
- L'operatore `equals(Object o)` invece guarda al contenuto che sta dentro alla memoria e quindi fa proprio un confronto fra oggetti

Il metodo *equals* deve essere riflessivo (`x.equals(x)` deve essere `true`), simmetrico (`x.equals(y)` è vero se e solo se `y.equals(x)`) e transitivo (se `x.equals(y) && y.equals(z)`) allora `x.equals(z)`). Per una corretta implementazione di `equals()` conviene:

```
public class ColorPoint {
      private Point point;
      private Color color;

      @Override
      public boolean equals(Object obj) {
           1 if(this == obj) return true;
           2 if(!(obj instanceof ColorPoint)) return false;
           3 ColorPoint cp = (ColorPoint)obj;
             return cp.point.equals(point) &&
                    cp.color.equals(color);
      }
```

In pratica si devono eseguire 3 step nella classe più un quarto di verifica all'esterno:

1. Usare == per vedere se l'argomento è uguale a this
2. Usare `instanceof` per vedere se l'argomento va bene oppure no
3. Castare l'argomento al tipo giusto e verificare ogni "campo" che ci interessa comparare
4. Controllare se stiamo rispettando riflessività, simmetria e transitività

Torna utile ricordare cosa fa l'operatore `instanceof` in Java: in pratica serve a verificare se un oggetto è una istanza di un determinato tipo (classe, sottoclasse o interfaccia). Confronta l'istanza con il tipo e ritorna vero o falso. Vediamo degli esempi:

```
class Simple1{
  public static void main(){
    Simple1 s = new Simple1();
    var x = s instanceof Simple1;
 }
}
```

Qui `x` è `true` perché s è un'istanza di Simple1; infatti il tipo di s è proprio Simple1.

```
class Animal{}
class Dog1 extends Animal {
  public static void main(){
    Dog1 d = new Dog1();
    var x = d instanceof Animal
  }
}
```

Qui `x` è `true` perché Dog è sottotipo di Animal quindi Dog al suo interno contiene il "pezzo" dell'oggetto Animal.

Quindi questo operatore lo uso per vedere se una variabile corrisponde a un tipo o se è un sottotipo di qualche altro tipo. Nell'esempio sopra se avessi scritto:

```
Dog1 d = null;
var x = d instanceof Animal;
```

In questo caso `x` sarebbe `false` perché d non è di tipo Animal e tantomeno è un sottotipo di Animal.

Solitamente è buona prassi fare l'override del metodo `toString()` che si trova in Object e ritorna una rappresentazione testuale dell'oggetto o il contenuto che esso contiene in formato stringa. Importante ricordare che:

```
String nome = new String("Roberto")

String nome = "Roberto"
```

L'operazione più efficiente a lungo andare è la seconda. Nel primo caso ho una reference perché alloco nello heap un oggetto, nel secondo caso **NON** ho reference e costruisco la stringa che è costante e so dove si trova in memoria

Inoltre, per quanto riguarda l'operazione di concatenazione di stringhe, è importante ricordare che.

```
String nomi = "";
for(var p : people)
  nomi += p;

var nomi = new StringBuilder("");
  for(var p : people)
    nomi.append(p);
```

Il tipo `String` è immutabile e ogni volta che si fa il += viene allocata una nuova stringa, quindi chiamare += spesso causa un continuo ri-alloco di stringhe. Lo `StringBuilder` invece viene allocato una volta sola (al momento di creazione, col new) e poi con l'append si aggiunge testo in coda senza le riallocazioni (come accadeva prima)

Infine, è da ricordare che bisogna evitare di scrivere questo tipo di codice. Compila, ma è inefficiente.

```
Long sum = 0L;
var max = Integer.MAX_VALUE;

for(long i = 0; i < max; ++i)
  sum += i;
```

Il tipo di sum è `Long`, ovvrero un wrapper di long. Nel for invece ho `long` che è un tipo primitivo e quindi la chiamata a `sum += i;` effettua il boxing del tipo, quindi crea conversioni di continuo che impattano sulla performance del programma. Soluzione: usare `long sum;`.

Con **autoboxing** si intende il processo di conversione dal tipo primitivo al rispettivo wrapper, come è accaduto sopra. Codice del tipo `Integer numero = 5;` fa autoboxing perché converte da solo il 5, che

è di tipo `int`, in un `Integer`. Un equivalente sarebbe `Integer numero = new Ingeter(5);`. Con **unboxing** si intende il processo contrario, cioè il passare da una classe wrapper al tipo primitivo che essa rappresenta. Considerando l'esempio di prima, fare `int valore = numero` è fare unboxing perché sto assegnando ad un `int` (valore) un `Integer` (numero). Quindi:

- Autoboxing: da int a Integer *(esempio)*
- Unboxing: da Integer a int *(esempio)*

Può tornare utile da usare il metodo della classe Object `clone()` che ritorna una copia profonda dell'oggetto. Per farlo si implementa l'interfaccia `Cloneable` e si richiama il metodo clone della superclasse; poi si fa un cast e si procede alla copia profonda dei metodi restanti.

```
public class Stack implements Cloneable {
   private Object[] elements;

   @Override public Stack clone() {
        try {
            Stack result = (Stack)super.clone();
            result.elements = elements.clone();
        }catch(CloneNotSupportedException cnsex)
            throw new AssertionError();
        }
        return results;
   }
}
```

Qui si vede che la prima cosa da fare è chiamare il metodo clone della superclasse e castare il risultato al tipo stesso della classe. Poi si fa una copia profonda dei campi che hanno delle reference mutabili e si ritorna il risultato. Notare che su elements è stato chiamato `clone()` perché è il metodo che fa una copia profonda di un array di `Object` ma volendo si sarebbe potuto fare manualmente un for e fare deep copy del contenuto dell'array.

La `clone()` comunque potrebbe essere un problema se un campo è `final`, in generale è meglio avere un metodo di cloning separato e non implementare `Cloneable`. Ad esempio, si può usare un costruttore di copia oppure un *factory method*, ovvero un metodo che costruisce per me l'oggetto. Tipo

```
public static Stack factoryNew(Stack source) {
      Stack dest = new Stack(...);
      //faccio copie deep o altro prendendo i dati da source
      return dest;
}
```

L'idea del factory è: creo un metodo da usare tipo `Stack copia = Stack.factoryNew(...)` che ritorna un oggetto di tipo Stack che è una copia esatta dell'oggetto che passo in input.

Esiste inoltre il metodo `finalize()` che si trova nella classe Object e viene chiamato dal garbage collector prima di togliere l'oggetto dallo heap. Non si può sapere quando il thread di pulizia passerà e quindi non possiamo avere il controllo su quando verrà chiamato il metodo. La sua esecuzione può impattare sulle performance.

Quando ci sono molti parametri in input può tornare utile il *Builder Pattern* che permette di costruire un

oggetto utilizzando un “helper”, ovvero una classe interna che fa da costruttore. Ecco un esempio.

```java
public class BuilderPattern {
    private final int arg1;
    private final int arg2;

    private int optional1;
    private int optional2;

    public static class Builder {
        private final int arg1;
        private final int arg2;

        private int optional1;
        private int optional2;

        public Builder(int a, int b) {
            this.arg1 = a;
            this.arg2 = b;
        }

        public Builder opt1(int value) {
            optional1 = value;
            return this;
        }

        public Builder opt2(int value) {
            optional2 = value;
            return this;
        }

        public BuilderPattern build() {
            return new
                   BuilderPattern(this);
        }
    }

    public void showVariables() {
        System.out.println(arg1);
        System.out.println(arg2);
        System.out.println(optional1);
        System.out.println(optional2);
    }

    public BuilderPattern(Builder bld) {
        arg1 = bld.arg1;
        arg2 = bld.arg2;

        optional1 = bld.optional1;
        optional2 = bld.optional2;
    }
}
```

```java
public static void main(String[] args) {
    var p = new BuilderPattern.Build(1, 5)
                              .opt1(3)
                              .build();
        p.showVariables();
        //stampa: 1 5 3 0
}
```

Intanto, il `var p = new BuilderPattern(...)` è come `BuilderPattern p = new BuilderPattern`.

La classe accetta due parametri in per forza e poi due sono opzionali. Senza scrivere manualmente tanti costruttori quante sono le variabili opzionali, si può usare il *Builder Pattern*.

In pratica si crea una classe interna (`Builder`) che ha gli stessi parametri di quella esterna e il suo costruttore prende i parametri obbligatori. Poi ci sono i metodi `optX(int)` che settano i valori opzionali e la parte fondamentale è `return this`. In questo modo ritorno sempre lo stesso oggetto e posso fare l’effetto “catena” (similmente a come si fa in C++ con l’overload di ostream). Alla fine il `build()` mi ritorna l’oggetto che volevo e gli passo `this`, ovvero la classe con i parametri che ho appena costruito.

Nel costruttore di `BuilderPattern` mi basta fare una semplice assegnazione 1:1 dei campi ed è fatta. Notare che `arg1`, `arg2` e gli altri sono tutti degli `int` e quindi vengono inizializzati automaticamente a 0 (i primitivi sono “managed types”).

Più avanti verranno discusse le classi interne però basta sapere, per ora, che una classe interna ha accesso a tutti i membri (anche privati) della classe esterna. Una classe interna **non** si può dichiarare da sola ma va sempre dichiarata con un’istanza del tipo esterno che la racchiude.

Una classe immutabile è una classe che, una volta creata, non può cambiare il suo contenuto (come ad esempio la classe `String`). Prima infatti abbiamo visto che la concatenazione col += di stringhe non è molto efficiente perché le stringhe, essendo immutabili, non si possono modificare e quindi il += crea una nuova stringa ogni volta.

Una classe immutabile generalmente si crea se non ha alcun membro esposto e non ha alcun setter; diciamo che è in “read only mode” e quindi in quanto tale è anche thread safe. Più avanti vedremo che i problemi avvengono quando più processi vogliono simultaneamente scrivere su una risorsa condivisa ma se tale risorsa è di sola lettura, ognuno può leggere quando vuole e quanto vuole. In generale, le regole da rispettare per creare una classe immutabile sono:

1. Non fornire alcun “mutatore” tipo setter o metodi pubblici
2. Bloccare l’override dei metodi la classe `final`
3. Mettere i metodi `final`

4. Mettere tutti i campi privati
5. Essere sicuri di avere accesso esclusivo ad ogni “componente mutabile”

Per chiarire il punto 5, vediamo questo pezzo di codice:

```
public class MyDateTime {

        private Date dateTime;
        private TimeZone tz;

        public MyDateTime(Date dateTime, TimeZone tz) {
                /** checks if class invariants are respected ...*/
                this.dateTime = dateTime;
                this.tz = tz;
        }

        public Date getDateTime() {return dateTime;}}
}
```

Non va bene qua perché sto inizializzando due classi (`dateTime` e `tz`) semplicemente facendo una copia di reference; **non** sto facendo una copia profonda ma sto solo copiando il puntatore. Inoltre sto ritornando una reference ad un campo oggetto privato della classe! In realtà, già solo con il costruttore sono in pericolo perché mi basta modificare l’oggetto da fuori la classe per modificarlo anche dentro dato che non ho fatto una copia profonda.

Generalmente si tende a favorire la composizione in favore dell’eredirarietà. L’ereditarierà e l’override di metodi può essere pericoloso e più difficile di quanto ci si aspetti, proprio come accade per il metodo `equals`. La cosiddetta tecnica di composizione ci può salvare da questi pericoli:

Ereditarietà

```
public class Test extends Another {
    //codice...
}
```

Composizione

```
public class Test {
    private Another a = ... ;
    //codice...
}
```

In pratica al posto di ereditare e fare override di metodi si usa un campo privato interno alla classe e si usa quell’oggetto all’interno di alcuni metodi che andranno definiti nella classe contenitore, in questo caso la `Test`. Vediamo un esempio dove si crea una classe che ne estende un’altra e ha lo scopo di contare quanti elementi sono stati inseriti

```
// This is very common, but broken
public class InstrumentedHashSet<E> extends HashSet<E>
  private int addCount = 0; // add() calls
  public InstrumentedHashSet() {}
  public InstrumentedHashSet(Collection<? extends E> c)
     { super(c); }
  public boolean add(E o) {
     addCount++; return super.add(o);
  }
  public boolean addAll(Collection<? extends E> c) {
     addCount += c.size(); return super.addAll(c);
  }
  // accessor method to get the count
  public int addCount() { return addCount; }
}
```

Internamente la classe `HashSet` implementa `addAll()` basandosi su `add()`; in altre parole, la classe `HashSet` dentro al metodo `addAll()` chiama `add()`. Facendo così il metodo `addCount()` ritorna un numero sbagliato!

- La chiamata `addAll()` incrementa correttamente la variabile `addCount` ma poi fa una chiamata a `super.addAll(c);` questo chiama il metodo `add()` che oltre a chiamare (giustamente) `super.add(c);` chiama anche `addCount++`! Quindi il conteggio totale viene sballato perché incremento `addCount` di `c.size()` e poi anche di 1 col ++ ma non dovrei

Vediamo dunque che in questo caso possiamo "*favor compisition over inheritance*" per risolvere il tutto:

```
// Note that an InstrumentedSet IS-A Set
public class InstrumentedSet<E> implements Set<E> {
    private final Set<E> s;
    private int addCount = 0;

    public InstrumentedSet (Set<E> s) { this.s = s}
    public boolean add(E o) {
        addCount++; return s.add(o); }
    public boolean addAll (Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    // forwarded methods from Set interface
}
```

Adesso il problema non c'è più perché, avendo un oggetto interno, posso usare questo e non ho bisogno di fare override o fare chiamate a super. Implemento solamente l'interfaccia e faccio override dei contratti dei metodi specificati da quell'interfaccia (`Set<E>`).

In Java se in una classe c'è almeno un metodo astratto, allora la classe deve essere marcata `abstract` per non avere errori a compile time. Può esserci una classe marcata abstract senza metodi astratti dentro ma comunque è astratta. Caratteristiche:

- Tipicamente definiscono solo le caratteristiche fondamentali della classe
- Possono avere metodi astratti (da concretizzare nelle sottoclassi) e/o metodi non astratti che saranno disponibili alle sottoclassi (se sono `protected` o `public`)
- Non si possono creare oggetti di classi astratte
- Ci possono essere dei costruttori astratti
- Una sottoclasse di una classe astratta si dice *concreta* quando fornisce implementazioni di tutti i metodi astratti di A. Se **non** lo fa, anche tale sottoclasse è astratta
- Non posso avere `final` e `abstract` assieme anche perché non avrebbe senso

Prima di Java 8 le interfacce potevano avere solo metodi astratti pubblici come noto ed era necessario che una classe definisse tutti i metodi dell'interfaccia. Ad esempio:

```
interface TestInterface {
  public void square(int a);
  public void show()
}
```

```
class Test implements TestInterface {
  public void square(int a) { ... }
  public void show() { ... }
}
```

Se in `Test` non avessi definito `square(int a)` o `show()` avrei avuto un errore di compilazione. In Java 8 l'esempio è ancora perfettamente valido ma si può fare anche questo:

```
interface TestInterface {
  //square() è implicitamente
  //public ed abstract
  void square(int a);

  default public void show(){
    System.out.print("i-test");
  }
}
```

```
class Test implements TestInterface {
  public void square(int a) { ... }

  public static void main(...) {
    Test d = new Test();
    d.square(5); //ok, esegue square
    d.show();    //stampa: i-test
  }
}
```

In pratica è possibile aggiungere la segnatura `default` davanti alla dichiarazione del metodo e dargli un'implementazione; così facendo tutte le classi che implementano quell'interfaccia avranno a disposizione quel metodo senza doverlo definire per forza.

Notare però che `square(int a)` **non** è default e quindi, solito, serve definire il contratto nella classe che implementa l'interfaccia. Tuttavia il metodo di default "sparisce" quando viene ridefinito all'interno della classe che lo implementa:

```
interface TestInterface {
  public void square(int a);

  default public void show(){
    System.out.print("i-test");
  }
}
```

```
class Test implements TestInterface {
  public void square(int a) { ... }

  public void show(){
    System.out.print("i-test-overr.");
  }

  public static void main(...) {
    ...
    d.show();  //stampa: i-test-overr.
  }
}
```

Qui vediamo che è stato fatto l'override di `show()` (che è comunque un metodo virtuale!) e quindi la classe mostra *i-test-overr.* in output. La regola è che la classe "vince sempre" perché se non appare nulla si usa il metodo default, altrimenti vale l'override. Può anche capitare questo:

```
interface TestInterface1 {
  default public void show(){
    System.out.print("test1");
  }
}

interface TestInterface2 {
  default public void show(){
    System.out.print("test2");
  }
}
```

```
class Test implements TestInterface1,
                      TestInterface2 {
  public void show(){
    TestInterface1.super.show();
    TestInterface2.super.show();
  }

  public static void main(...) {
    ...
    d.show();  //stampa: i-test-overr.
  }
}
```

In questo caso c'è il problema che due interfacce diverse danno la stessa definizione del metodo show; il problema nasce perché `Test` implementa da entrambe le interfacce. Si crea un conflitto dato che Java non sa fra quale dei due scegliere. Per risolvere è necessario fare l'override e specificare cosa fare; in particolare la sintassi `NomeInterfaccia.super.metodo()` richiama il relativo metodo specificato di default. Un'interfaccia può anche avere un metodo statico ed ha le stesse regole dei normali metodi statici:

```
interface TestInterface {
  static void show(){
    System.out.print("test");
  }
}
```

```
public static void main {
  TestInterface.show();
  //stampa: test
}
```

Definire un metodo statico in un'interfaccia è uguale al definire un metodo statico all'interno di una classe; come per il `default` si può dare una definizione. La differenza con le classi astratte è che queste ultime possono avere un costruttore, uno stato e dei comportamenti.

In Java una classe (detta *outer class*) può contenere un'altra classe al suo interno (detta *inner class*). Le classi interne sono locali alla classe che le contiene ed esse **NON** possono essere istanziate direttamente. In pratica non posso fare la `new` di una inner, ma posso farlo tramite l'istanza di un oggetto outer. Esempio:

```
class Outer {
  private int a;
  public Outer(int value) {
      a = value;
  }

  public class Inner {
      private int b;
      public Inner(int value) {
          b = a + value;
      }
      public void showValue() {
          System.out.println(b);
      }
  }

  public void printValue() {
      System.out.println(a);
  }
}
```

```
public class Main {

  public static void main(...) {
    Outer o = new Outer(10);
    o.printValue(); //1

    Outer.Inner i = o.new Inner(20);
    i.showValue(); //2
  }
}
```

1. Stampa 10 perché chiama il metodo normalmente
2. Stampa 30.

Notare che `Inner i = new Inner(20)` non va bene perché le classi interne possono essere create **solo** se legate ad istanze di oggetti. Occorre quindi creare prima un oggetto della classe outer (fatto prima con `new Outer(10)`) e poi usare la sintassi `Outer.Inner i = o.new Inner(20)`

Utile anche notare che una classe interna ha accesso anche a ciò che è privato nella classe outer (variabili e metodi), infatti `Inner` accede ad `a` che è privata in `Outer`. Una classe outer non può accedere ai campi interni di una classe inner direttamente ma può accedere ai suoi campi se ne crea un'istanza.

- Se la classe interna non ha il metodo che è stato invocato, si assume che si sia nella outer
- Se c'è un conflitto di nomi, in caso di istanziazione di classe inner si usa il metodo della inner
- Se la classe inner e la classe outer hanno un metodo con lo stesso nome e la inner vuole chiamare quel metodo di outer allora si usa `OuterClassName.this.methodName();`

Se si dichiara una classe interna come `static` allora la si può accedere più facilmente come ad esempio `Outer.Inner i = new Outer.Inner(20);` e in questo caso `i` può chiamare metodi statici e non della inner e metodi statici della outer. Se una classe eredita da una outer, la sottoclasse ha accesso alla classe inner regolarmente ed eredita tutto (se quest'ultima è `protected` o `public` ovviamente)

Si possono anche creare le *anonymous classes* ovvero delle classi create "al volo" :

```
public class E {
    public I i() {
        return new I() { // annonymous inner class
                private int i = 5;
                public int value() { return i; }
        }; // Note the ";"
    }
    public static void main(String[] args) {
        E e = new E();
        I ref = e.i();
        ref.value();
    }
}
```

```
interface I {
  int value();
}
```

La definizione della classe è integrata nell'espressione di return o di dichiarazione di variabile, basta che ci sia l'operatore `new`. Una classe anonima è una notazione abbreviata per creare un semplice oggetto piccolo "inline" che racchiude del codice da eseguire. Di solito il tipo è una interfaccia. Nell'esempio sopra la classe è castata ad una interface; sarebbe equivalente a:

```
class Anonima implements I { ... };
return new Anonima();
```

Importante ricordare che una classe interna ha dei costruttori creati dal compilatore automaticamente di tipo `Inner(Outer)` e `Inner(Outer, Outer.Inner)`. Ad esempio:

```
public class Outer {
  private int x = 1;

  private Inner {
          private int x = 2;

          private InnerInner {
                  private int x = 3;
          }
  }
}
```

Per `InnerInner` il compilatore crea due costruttori:

- `InnerInner(Inner)`
- `InnerInner(Inner, Inner.InnerInner)`

Nel secondo caso, il primo parametro è il `this` della classe esterna mentre il secondo parametro è il `this` solito che fa riferimento al contesto di invocazione

Alcune note finali da ricordare riguardo le classi interne che possono tornare utili:

- La classe outer può accedere a tutti i membri (anche privati) della classe inner dopo avere istanziato il tipo tramite un oggetto
- Se una outer ha due classi interne, esse possono accedere ai loro membri privati a piacere!

In java è possibile creare classi e metodi **generici** grazie all'utilizzo della sintassi `ClassName<T>`.

```
public class ArrayList<T>
extends AbstractList<T>
implements List<T>, RandomAccess, Cloneable, Serializable {
  public boolean add(T obj) {...}
  public T elementAt(int index) {...}
  ...
}

ArrayList<String> v = new ArrayList<>();
ArrayList<Dipendente> w = new ArrayList<Dipendente>(5);
v.add(new String("pippo");
String s = v.elementAt(0);
// w.add(new String("pippo")); // NON COMPILA
```

Qui vediamo che la classe `ArrayList` è generica perché accetta un parametro chiamato T e questo verrà usato al suo interno per definire metodi e variabili. Come si può vedere, se `T` fosse di tipo `String`, allora i sarebbe un "tipaggio forte" e nessun altro tipo sarebbe accettato.

In pratica, una volta definito il tipo del parametro, bisognerà sempre tenere presente che verrà usato quel tipo. Se la classe sopra, cioè `Dipendente`, fosse una superclasse, non sarebbe comunque accettabile fornire una sua sottoclasse al metodo `add()`. Non centra l'ereditarietà, bisogna che il tipo sia uguale.

```
public class Coppia<T, S> {
  private T first;
  private S second;
  public Coppia(T x,S y) {first = x; second = y;}
  public T getFirst() {return first;}
  public S getSecond() {return second;}
}
```

Ecco un esempio di classe con due parametri generici. Quando si istanzia il tipo, basta fornire i tipi al posto dei parametri e il compilatore dedurrà automaticamente i tipi da sostituire alle variabili. Quindi:

```
var c = new Coppia<Integer, Double>(3, 7)();
```

Importante notare che i parametri generici possono essere istanziati a classi e interfacce ma **NON** a tipi primitivi. Per questo motivo è stata usata la classe wrapper `Integer` al posto di un `int` e nel costurttore, grazie al boxing del tipo, è stato possibile passare un `int` all'`Integer`. Notare che questo codice non compila:

```
public static <T> T min(T[] a) {
  T x = a[0];
  for(int i=1; i<a.length; ++i) if(a[i] < x) x=a[i];
    return x;
}
```

Il compilatore non può applicare l'operatore < perché non sa nulla su T, sa solamente che è un tipo che verrà istanziato. Non può sapere l'effetto dell'operatore sul tipo T perché non può sapere cosa diventerà T (`Integer`, `Double`, una classe, un'interfaccia...). Per risolvere il problema si può fare così:

```
public static <T extends Comparable<T>> E min(E[] a) {
  E x = a[0];
  for(int i=1; i<a.length; ++i) if(a[i].compareTo(x) < 0) x=a[i];
    return x;
}
```

In questo modo ho indicato il **vincolo** che T deve essere un sottotipo di `Comparable<T>`; così facendo posso usare il metodo `compareTo()` che è dichiarato in Comparable. Il compilatore sa che `compareTo()` lo può chiamare su T (indipendentemente da che tipo abbia) perché l'implementazione dell'interfaccia garantisce la presenza di quel metodo. Si può anche fare così:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

In questo modo T estende più classi e interfacce (prima vanno dichiarate le classi, poi le interfacce).

```
class Dirigente extends Dipendente {...}
class Agente extends Dipendente {...}

ArrayList<Dirigente> a = new ArrayList<>();
// se fosse possibile...
ArrayList<Dipendente> b = a;
Dipendente dip = new Agente();
b.add(dip); // ...non andrebbe bene!!
```

Anche se `Dirigente` è una sotto classe di `Dipendente` **NON** è vero che `ArrayList<Dirigente>` è una sottoclasse di `ArrayList<Dipendente>`. Il legame di ereditarietà fra i parametri di tipo **NON** genera un legame di ereditarietà tra le classi corrispondenti!

Quindi dati due tipi A e B allora `ClasseX<A>` non è in relazione con `ClasseX<B>` anche se A e B sono in relazione di sottotipo (A superclasse e B sottoclasse). In Java i Generics sono **invarianti** (l'ereditarietà non ha influenza) ma gli array sono **covarianti** (l'ereditarietà ha influenza).

Vediamo meglio il discorso di `<T extends Type>` cosa sta a significare di preciso nell'ambito Generics:

NOME	SINTASSI	SIGNIFICATO
Vincolo di limite superiore	`? extends B`	Qualsiasi sotto-tipo di B
Vincolo di limite inferiore	`? super B`	Qualsiasi super-tipo di B
Nessun vincolo di limite	`?`	Qualsiasi tipo (super e sotto)

Consideriamo il frammento di codice qui sotto ricavato dal package java.lang.Numbers per fare un esempio di utilizzo delle wildcard nei Generics.

```
public abstract class Number extends Object implements Serializable {…}
public final class Float extends Number implements Comparable<Float> {…}
public final class Double extends Number implements Comparable<Float> {…}
public final class Integer extends Number implements Comparable<Float> {…}
```

- `? extends B`

  Si usa il vincolo di limite superiore quando si vuole che il tipo T da mettere dentro alle parentesi del tipo generico (Class<**T**>) sia una sottoclasse di B. Vediamo un esempio:

```
public static double sumElementiLista(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

  Il metodo accetta il parametro `List<? extends Number>` e quindi posso passare ad esempio una cosa tipo `List<Float>` o `List<Double>`. Infatti Float e Double sono sottoclassi di Number e quindi posso chiamare quel metodo.

- `? super B`

  Si usa il vincolo di limite inferiore quando si vuole che il tipo T da mettere dentro alle parentesi del tipo generico (Class<**T**>) sia una superclasse di B. Vediamo un esempio:

```
public static void copy(Integer[] src, List<? super Integer> dst)
{
   Objects.requireNonNull(src, "Source should not be null");
   Objects.requireNonNull(dst, "Destionation should not be null");
   for (Integer i: src) {
        dst.add(i);
    }
}
```

  Il metodo accetta il parametro `List<? super Number>` e quindi posso passare qualcosa che sia superclasse di `Integer` come ad esempio `List<Number>`. Infatti `Integer` deriva da `Number` e quindi va bene.

- `?`

  Si usa quando va bene qualsiasi tipo T dentro alle parentesi del tipo generico (Class<**T**>)

Quando uso `extends T` sto dicendo che voglio un parametro che sia T o inferiore (una sottoclasse) e quando uso `super T` sto dicendo che voglio un parametro che sia T o superiore (una superclasse).

Notare che per implementare i Generics il compilatore di Java applica la cancellazione di tipo (*type erasure*), cioè sostituisce tutti i parametri nei tipi generici con Object o col primo tipo "disponibile" se si usano relazioni come ad esempio `extends`. Il bytecode contiene solo classi ordinarie, interfacce e metodi.

```
public class Coppia<F, S> {
  private F first;
  public Coppia(F x, S y) {
    first = x;
  }
  public F getFirst() {return first;}
}
```

```
class MioOggetto<T extends Comparable<T>> {
    private T data;
    public MioOggetto(T data) {
        this.data = data;
    }
    public int compareTo(T o) {...}
}
```

. . . viene tradotto in . . .

. . . viene tradotto in . . .

```
public class Coppia {
  private Object first;
  public Coppia(Object x,Object y) {
    first = x;
  }
  public Object getFirst() {
    return first;
  }
}
```

```
class MioOggetto {
    private Comparable data;
    public MioOggetto(Comparable data) {
        this.data = data;
    }
    public int compareTo(T o) {...}
}
```

Vediamo infatti che dove non sono state specificate "restrizioni" viene tutto convertito ad Object, mentre dove c'è la restrizione di limite superiore viene applicato il tipo più "alto" possibile. Si cerca sempre di avere Object o comunque il suo derivato più prossimo.

Tutte le verifiche di tipo sono fatte a tempo di compilazione e non c'è nessun onere a run-time. Tuttavia nei generici non è possibile:

1. Non si possono creare istanze di un parametro di tipo; cioè `T test = new T();` non è valido. In generale, tutte le azioni che richiedono informazioni a run-time sul tipo non sono concesse.
2. Non si possono usare i parametri di tipo di una classe generica per definire campi, metodi o classi **statiche** come ad esempio:

```
public class List<T> {
  private static T defaultValue; // ILLEGALE
  public static List<T> replicate(T value, int n) {...} // ILLEGALE
  private static class Nodo {
    public T info; // ILLEGALE
    public Nodo next;
  }
}
```

Tuttavia per i metodi statici e le classi interne basta aggiungere un tipo parametrico all'inizio:

```
public class List<T> {
public static <T> List<T> replicate(T value, int n) {...} // OK
  private static class Nodo<T> {
    public T info; // OK
    public Nodo<T> next;
  }
}
```

3. L'overload di metodi che differiscono solo da un tipo generico produce un errore di compilazione:

```
public class UsoLista<W, T> {
        void operazioneLista(List<W> v) {…}
        void operazioneLista(List<T> v) {…}
}
```

È meglio usare `List<T>` invece di `List` (che si basa su `Object`) ed è meglio preferire le liste agli array di solito. I generics sono fondamentali per lo sviluppo delle collezioni. Una collezione è un singolo oggetto che rappresenta un gruppo di oggetti; mentre gli array di Java sono creati con una lunghezza fissata, un oggetto di tipo `Collection` è un array dinamico di `Object`. `List<T>` è una classe **astratta**.

Importante è l'interfaccia `List<T>` che rappresenta una sequenza di elementi e le principali classi concrete che vengono utilizzare spesso nel codice sono:

- ArrayList

  ```
  public class ArrayList<E> extends AbstractList<E> implements
  List<E>, RandomAccess, Cloneable, Serializable
  ```

  Si tratta di una struttura dati efficiente in accesso O(1), lenta in rimozione O(n) di elementi e mediamente efficace nell'inserimento (O(1) ammort. / O(n) worst case; se non deve essere aumentata la dimensione dell'array interno). All'interno è implementata come un array che viene dinamicamente "re-sized".

- LinkedList

  ```
  public class LinkedList<E> extends AbstractSequentialList<E>
  implements List<E>, Deque<E>, Cloneable, Serializable
  ```

  Si tratta di una struttura efficiente in inserimento O(1) e cancellazione però, generalmente, è lenta nell'accesso casuale. Essendo una lista doppiamente linkata, si scorrono uno ad uno i puntatori ai nodi prima di arrivare all'elemento quindi è una ricerca lineare in O(n).

Diciamo che `ArrayList<T>` di Java è come `std::vector<T>` di C++ mentre `LinkedList<T>` di Java è come `std::list<T>` di C++. La prima è per accesso casuale, la seconda per sequenziale. Ancora, similmente a C++, c'è la *capacity* in ArrayList che dice quanti spazi sono allocati ed è sempre >= della *size*.

Entrambe **NON SONO** thread safe e hanno gli stessi metodi ma `LinkedList<T>` ne ha qualcuno in più per modellare strutture specifiche come `Queue` e `Stack`.

Importante è anche l'interfaccia `Map<K, V>` che rappresenta un array associativo in quanto associa un oggetto ad un altro stabilendo una relazione (chiave, valore). Non ci possono essere duplicati.

- HashMap

  ```
  public class HashMap<K,V> extends AbstractMap<K,V> implements
  Map<K,V>, Cloneable, Serializable
  ```

  Si tratta di un insieme **non ordinato** di elementi in forma (chiave, valore) ed è implementata tramite una hash table ed utilizza una funzione hash per ricavare la chiave. In caso di collisione di chiavi, si usa il chaining. Complesità media è O(1) per le operazioni di get / put / delete.

- TreeMap

```
public class TreeMap<K,V> extends AbstractMap<K,V> implements
NavigableMap<K,V>, Cloneable, Serializable
```

Si tratta di un insieme **ordinato** di elementi in forma (chiave, valore) ed è implementato tramite un albero rosso-nero e quindi ne eredita le performance (ovvero O(logn)) per ricerca, presenza, inserimento e rimozione.

Ancora, nessuna di queste è thread-safe. Di default una `HashMap<T>` ha all'interno una tabella di 16 posti e un fattore di carico di 0.75; vediamo un po' nel dettaglio come è fatta:

Bucket	Entries List
0	Entry1 | Entry0
1	null
2	Entry0
...	...
15	Entry0

Per individuare la posizione di un elemento all'interno della struttura gli step sono:

1. Calcolare l'`hash` partendo dal valore da inserire. **Importante** non fare side effect sull'oggetto inserito perché il suo hash potrebbe cambiare!
2. Il calcolo della posizione `(hash % (tableLength-1)` sarà la chiave (l'indice dell'array in cui mettere il dato)

Notare come, in caso di collisioni di bucket, viene applicato il metodo standard di chaining tramite una lista concatenata. Si fa una ricerca lineare (in tempo O(n) quindi) dentro alla entries list per individuare il valore *value* mappato dalla chiave *key*. Vediamo come:

```
for(var entry : bucket) {
      if ((entry.key.hashCode() == key.hashCode()) &&
            ((key == entry.key) || key.equals(entry.key)))
            return entry.value;
      else
            return null;
}
```

Ottengo la posizione in tabella (il bucket) con hash % tableLength – 1. Poi verifico che quello che sto richiedendo sia nella lista di entries; per farlo, controllo che siano uguali gli hash codes e controllo che gli oggetti siano uguali, usando **equals**.

Appare chiaro come sia importante il fatto che se si va a fare override di `equals(Object obj)` sia importante fare l'override anche del metodo `hashCode()`. Quest'ultimo è utilizzato da certe strutture ed è fondamentale! Una buona implementazione per hashCode, quando si fa override di equals, solitamente è data da:

```
public int hashCode() {
    return Objects.hash( ... params list ...);
}
```

Se due oggetti sono uguali, allora **bisogna** che la chiamata al metodo `hashCode()` nei due oggetti ritorni lo stesso risultato. Di default la `hashCode()` definita in Object ritorna interi non basandosi sui valori delle variabili ma su altro. Quindi se ridefinisco il metodo `equals` usando variabili interne alla classe o altri valori, allora devo adattare anche il metodo `hashCode()` usando quelle variabili.

Infine è da notare anche l'interfaccia `Set<K, V>` che rappresenta una collezione di elementi nella quale non ci possono essere duplicati. Ad esempio viene spesso usato come struttura di ritorno da una HashMap per avere una lista delle chiavi.

- HashSet

```
public class HashSet<K> extends AbstractSet<K> implements Set<K>,
Cloneable, Serializable
```

Implementa l'interfaccia Set e si appoggia su una HashMap senza dare garanzie sull'ordine degli elementi dentro all'insieme (permette il null). Operazione di aggiunta / rimozione / presenza elemento è O(1).

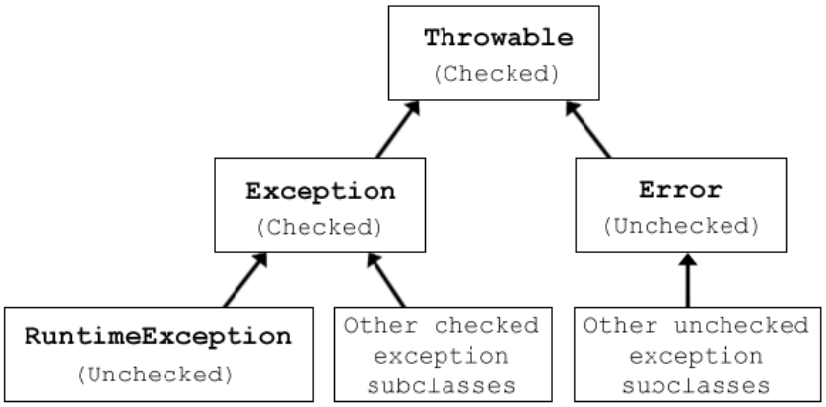
- TreeSet

```
public class TreeSet<K> extends AbstractSet<K> implements
NavigableSet<K>, Cloneable, Serializable
```

Implementa l'interfaccia Set e garantisce un insieme **ordinato** di elementi in forma basandosi sull'ordinamento naturale. La sua implementazione si basa su un albero rosso-nero garantendo il costo O(logn) nelle principali operazioni.

Nessuna di queste è thread-safe. Infine, ricordare la classe `Arrays` che fornisce metodi statici che operano su strutture lineari come array e liste.

Torna utile fare un breve ripasso sulle eccezioni in Java ed è utile ricordare quale sia la gerarchia definita dal linguaggio, dove `Throwable` ovviamente è sottoclasse di `Object`:



- `Error` e sottoclassi non sono catturate sotto condizioni normali e riguardano la JVM, non sono gestite dal programmatore e non sono "catturabili". Sotto ad essa ci sono altre sottoclassi (tante) non mostrate nel disegno
- `Exception` sono le eccezioni che il programmatore può catturare. Fra le sue tante sottoclassi c'è `RuntimeException` che (con le sue sottoclassi) è importante perché da molte famose definizioni di errori.

Se viene lanciata un'eccezione ti tipo `Error`, `RuntimeException` o una loro sottoclasse si dichiara solo il metodo tipo public `void test(){ }` altrimenti bisogna usare questo costrutto: `void test() throws ExceptionName{ }`. `Throwable`, `Exception` e le sue sottoclassi (tranne il ramo `RuntimeException`) sono eccezioni **checked**, ovvero che vanno controllate dal compilatore (serve il quindi specificare il `throws` nella segnatura del metodo o comunque un try-catch per gestirle).

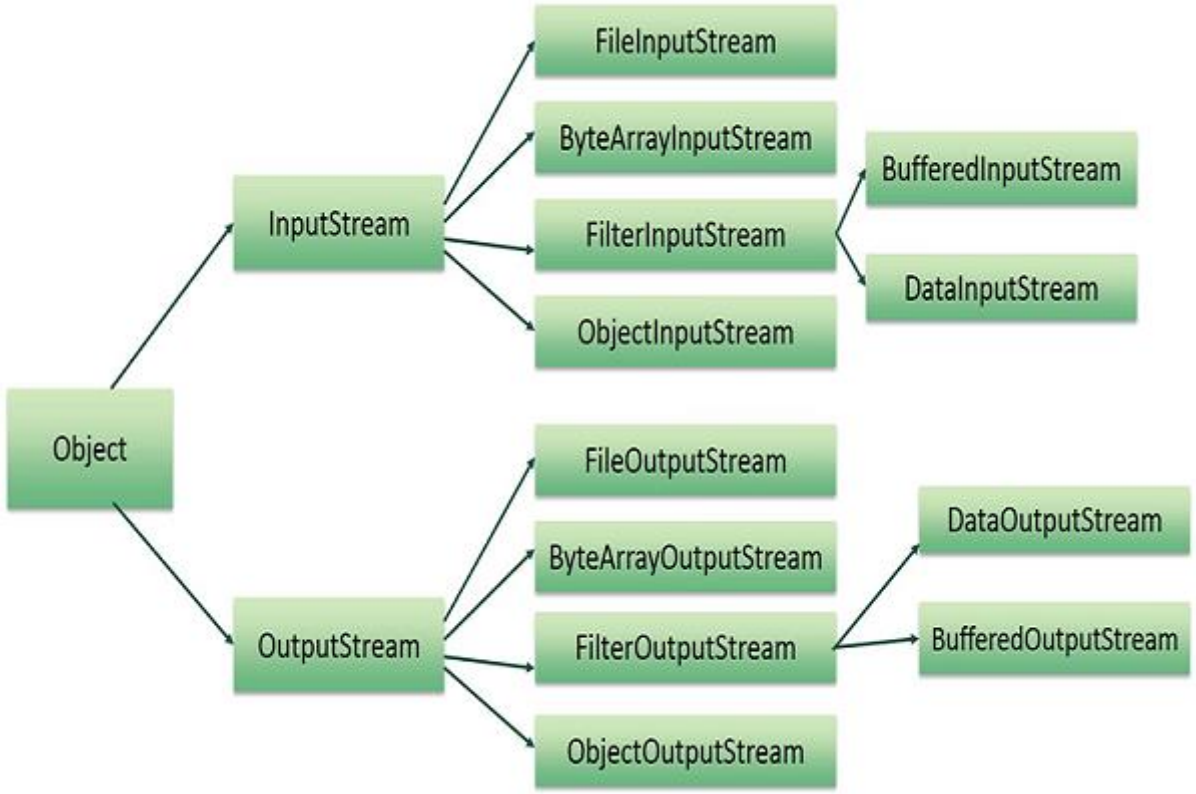
Si possono avere più catch in serie ma attenzione perché se si fa il catch di una classe e poi subito dopo il catch di una sottoclasse si ha errore perché l'eccezione è già stata gestita. Inoltre in Java 7 è stato introdotto il try-with-resources:

```
var s = new FileInputStream("a");
try {
  //... codice
} finally {
  if (streamObject != null) {
    streamObject.close()
  }
}
```

```
try(FileInputStream s = new
    FileInputStream("a")) {
  //..codice
}
```

In pratica con questo costrutto è equivalente al codice di sinistra; sto usando un try finally ed alla fine libero la risorsa chiamando close().

Uno stream è una sorgente o una destinazione di byte e `java.io` è divisa in due gerarchie di classi importanti a seconda del dato su sui operano: stream su byte o stream su carattere. Di entrambi ovviamente si può fare input ed output.



Stream su byte.

Vediamo che le due classi principali sono `InputStream` ed `OutputStream` con tutte le sottoclassi; vediamo alcuni dettagli implementativi:

- InputStream

  ```
  public abstract class InputStream extends Object implements
  Closable, AutoClosable
  ```

  Superclasse che rappresenta un input stream di byte e ha metodi bloccanti che quindi necessitano di paralellizzazione manuale (su thread) se desiderata. Notare che implementa anche `AutoCloseable` e quindi si può usare il try-with-resources.

- OutputStream

  ```
  public abstract class OutputStream extends Object implements
  Closeable, Flushable, AutoCloseable
  ```

  Superclasse che rappresenta un output stream di byte e ha metodi bloccanti che quindi necessitano di paralellizzazione manuale (su thread) se desiderata. Anche qui c'è `AutoCloseable` implementata.

Poi ci sono le classi di stream per la lettura/scrittura da e su dati presenti in memoria:

- `ByteArrayInputStream(byte[] buff)`: creato su un array esistente, legge i vari bytes e usa un contatore interno per tenere conto di dove è arrivato

- `ByteArrayOutputStream()`: i dati sono scritti in un array di byte ed il buffer aumenta automaticamente di dimensione dopo che i dati sono stati scritti e di possono recuperare (dall'oggetto) in foramato array di byte o stringa.

Non è necessario rilasciare la risorsa col metodo `close()` perché è tutto in-memory. Un esempio:

```
byte[] source = ...;
ByteArrayInputStream bis = new ByteArrayInputStream(source);
// leggi byte da sorgente...

ByteArrayOutputStream bos = new ByteArrayOutputStream();
// scrivi dei byte, output usando anche toString()...
byte[] sink = bos.toByteArray();
```

Le classi `FileInputStream` e `FileOutputStream` lavorano su uno stream di byte per input e output su file di qualsiasi filesystem. Viene aperta una connessione col file locale e se viene aperto si possono usare i vari metodi tipo `write()`, `read()` (in base se FIStream o FOStream) e il close.

Le classi `BufferedInputStream` e `BufferedOutputStream` permettono la bufferizzazione di dati evitando che ogni operazione di read e write acceda allo stream. Così facendo evito accessi su disco e pesco i dati da quel buffer che agisce da "proxy". Se il `BufferedInputStream` è vuoto si fa una lettura da stream quindi si pesca il dato da disco, lo si mette nel buffer e poi si prende da li.

Stream su caratteri.

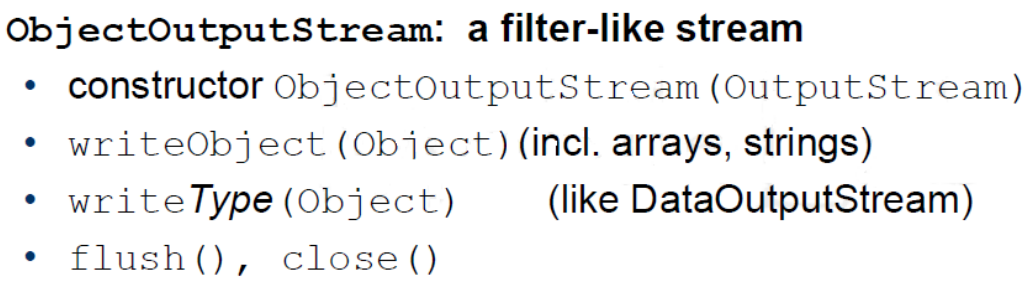
Gli stream che abbiamo appena visto leggono e scrivono i byte "grezzi", cioè in formato binario e ritornano la rappresentazione intera di quel binario. Le classi astratte `Reader` e `Writer` permettono di elaborare i dati grezzi e mostrarli come caratteri facendo una rielaborazione.

- Gli stream di byte non gestiscono bene i caratteri Unicode a 16 bit; fondamentali sono le due classi `InputStreamReader` e `OutputStreamWriter` che si usano come "interfaccia" per leggere i flussi di byte e convertirli in caratteri (e vice versa) secondo una certa codifica.
- Come per l'IO la conversione dei reader/writer è più efficace se si usa un buffer

Un concetto importante è quello della **serializzazione** (e la sua controparte, cioè la **deserializzazione**) che avviene in Java sia nativamente che non. In termini pratici e semplici si tratta di:

- *Serialization*: prendere un'oggetto Java e convertirlo in una qualche rappresentazione salvabile su di un file. Ad esempio, posso prendere un'oggetto e rappresentarlo con JSON in un file di testo
- *Deserialization*: il processo contrario a quello sopra, cioè ricavare un oggetto Java partendo da una fonte testuale o binaria. Ad esempio, dato un formato JSON su un file di testo posso (facendo il parse del file) ricavare un oggetto

Java offre un meccanismo built-in di serializzazione grazie all'interfaccia `Serializable`; essa va implementata se si vuole fare serializzazione/deserializzazione. Non contiene alcun metodo, serve solo da "marker". Ci sono due classi per fare serializzazione (`ObjectOutputStream`) e deserializzazione (`ObjectInputStream`), cioè per passare rispettivamente oggetto → file e file → oggetto.



`ObjectInputStream`: a filter-like stream

- constructor `ObjectInputStream(InStream)`
- `readObject()` (incl. arrays, strings)
- `read`*`Type`*`()` (like DataInputStream)
- `close()`

Per la serializzazione è necessario implementare l'interfaccia `Serializable` ed è consigliato fornire un ID univoco di tipo `long` che andrà ad indentificare la classe. Se non ci dovesse essere, ne verrà creato uno automaticamente.

```
public class MyObject implements Serializable {

        private static final long serialVersionUID = -2412988445012479811L;
        private int value;

        public MyObject(int value) { this.value = value;}

        public int getValue() { return value;}

        public static final void main(String args[]) throws IOException {

                try(ObjectOutputStream writer = new ObjectOutputStream(new
                        FileOutputStream("myObj.bin"))) {

                        writer.writeObject(new MyObject(4));
                }
        }
}
```

I campi statici **non** vengono catturati. Alla fine si otterà un file chiamato `myObj.bin` che conterrà una serie di byte the rappresentano la classe `MyObject` coi suoi campi. Vediamo il processo inverso di deserializzazione che dal file .bin crea la classe.

```
public class MyObject implements Serializable {

        private static final long serialVersionUID = -2412988445012479811L;
        private int value;

        public MyObject(int value) { this.value = value;}

        public int getValue() { return value;}

        public static final void main(String args[]) throws IOException {

                try(ObjectInputStream reader = new ObjectInputStream(new
                        FileInputStream("myObj.bin"))) {

                        MyObject myObj = null;
                        try {
                                myObj = (MyObject)reader.readObject();
                        } catch (ClassNotFoundException e) {
                                e.printStackTrace();
                        }
                System.out.println(myObj.getValue());//prints 4;
        }}
}
```

Notare l'utilizzo del *try-with-resources* che automaticamente chiama il `close()` sullo stream dopo averne terminato l'utilizzo. Durante la deserializzazione viene chiamato il costruttore di default (senza parametri)

della prima superclasse che **NON** implementa `Serializable`. Serve quindi essere sicuri che ci sia un costruttore senza parametri in qualche superclasse della gerarchia affinchè si possa usare questa feature. Bisogna notare che però questo metodo nativo di serializzazione ha diversi contro:

1. Diminuisce la flessibilità della classe una volta che è stata rilasciata. Se per qualche motivo dopo un paio di mesi dovessi cambiare dei campi interni (tipo aggiungere o togliere) avrei problemi perché l'UID potrebbe non matchare. Anche se lo facessi matchare, comunque avrei campi in più che nel file creato sulla classe vecchia non ci sono!
2. Non c'è alcun costruttore esplicito associato con la deserializzazione quindi bisogna essere **SICURI** tramite dei metodi che le invarianti della classe siano definite. Ad esempio una classe che mi dice gli appuntamenti futuri non può avere una data minore di quella di oggi! Devo fare dei controlli nei metodi per garantire la veridicità
3. Ci sono i metodi `writeObject()` e `readObject()` che, se definiti all'interno di una classe che implementa `Serializable`, vengono utilizzati per serializzare/deserializzare i dati secondo le specifiche date da quel metodo (invece di fare tutto di default). L'implementazione di default non sempre va bene e bisogna stare attenti.

Se non vogliamo che un campo non venga serializzato basta usare la keyword `transient` e in questo modo esso verrà ignorato.

```
public class MyClass implements Serializable {
      public transient Thread myThread;
      private transient String s;
      private int i;
}
```

Non verranno messi nel file i campi `myThread` ed `s` ma soltanto `i`. Vediamo una ridefinizione dei metodi per la lettura e scrittura da serializer:

```
public final class StringList implements Serializable {
         private transient int size = 0;
         private transient Entry head = null;
         // Non piu' serializzabile
         private static class Entry {
                  String data;
                  Entry next;
                  Entry previous;
         }

         public final void add(String s) { ... }

         private void writeObject(ObjectOutputStream s) throws IOException {
                  s.writeInt(size);
                  // Scrive tutti gli elementi in ordine
                  for (Entry e = head; e != null; e = e.next)
                  s.writeObject(e.data);
         }
         private void readObject(ObjectInputStream s) throws IOException,
         ClassNotFoundException {
                  int numElements = s.readInt();
                  // Legge tutti gli elementi e riscostruisce la lista
                  for (int i = 0; i < numElements; i++)
                           add((String) s.readObject());
}}
```

Perché usare i due metodi? Per esempio, se si dovesse avere un `Socket` quest'ultimo non è serializzabile e

quindi bisogna definire dei metodi per poter salvare le informazioni di tale socket. Se la classe è "semplice" cioè devo scrivere dati primitivi o Stringhe ok, però per tipi di dato complessi diventa complicato. Vediamo un esempio:

```
public class A implements Serializable {

 private transient Socket socket;

 private void writeObject(ObjectOutputStream out)throws IOException {
    out.defaultWriteObject();

    // prendo porta e IP dal socket e li butto fuori nel file
    InetAddress inetAddress = socket.getInetAddress();
    int port = socket.getPort();
    out.writeObject(inetAddress);
    out.writeObject(port);
 }

 private void readObject(ObjectInputStream in)
                  throws IOException, ClassNotFoundException{
    in.defaultReadObject();
    // leggo porta e IP dal file e li assegno all'oggetto socket che ho
    InetAddress inetAddress = (InetAddress) in.readObject();
    int port = in.readInt();
    socket = new Socket(inetAddress, port);
 }

}
```

Qui `Socket socket` è stato messo `transient` così che la classe `A` possa essere serializzata. Vediamo che i due metodi servono a dare delle specifiche su come serializzare la classe ed è necessario: la serializzazione di default (che fa Java, cioè con i `writeObject` e `readObject` di default) non farebbe così. Ridefinire i due metodi quindi viene utile quando ho certe cose da prendere in considerazione.

- Serializzazione binaria: traduco tutto in binario su di un file. È efficiente e veloce da fare il parse però non è leggibile dall'uomo ed è legato alla piattaforma
- Serializzazione cross-platform: ad esempio si fa con XML (che in caso di grandi oggetti diventa illeggibile) oppure con JSON (che va molto perché è leggero e facile da fare il parse).
- Serializzazione con Protobuf: libreria realizzata da Google che è molto efficiente, efficace e supera di gran lunga XML. Si tratta di un modo fra i tanti di fare serializzazione ma è molto veloce

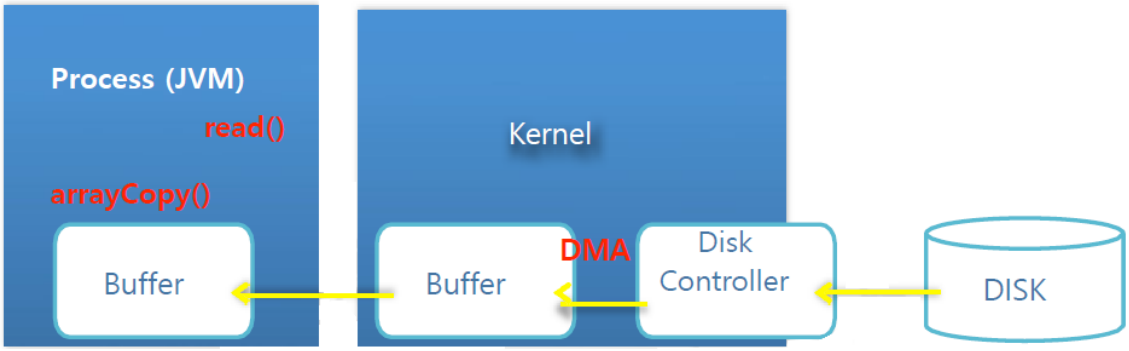
Java NIO

Vediamo ora l'argomento dell'input/output di `java.nio` e `java.nio.2`; sono entrambi utilizzati per fare input ed output. Le principali differenze riassunte sono:

- `java.io` era basato sulla "metafora" dello stream tradizionale, cioè input di byte/output di byte. Non c'era alcun modo di accedere a dati da multiple risorse senza incappare nel context switch dei thread. Inoltre, zero supporto per i "trick" che ormai usano i SO moderni per I/O ad altre prestazioni (come *memory mapped files*).
- `java.nio` crea una gerarchia di *buffer classes* che permette di spostare i dati dalla JVM al SO con un numero minimo di copie in memoria. Introduce i *channels* che permettono ai dati di passare direttamente dai buffer ai file/socket senza passare per gli stream (che facevano da intermedi).

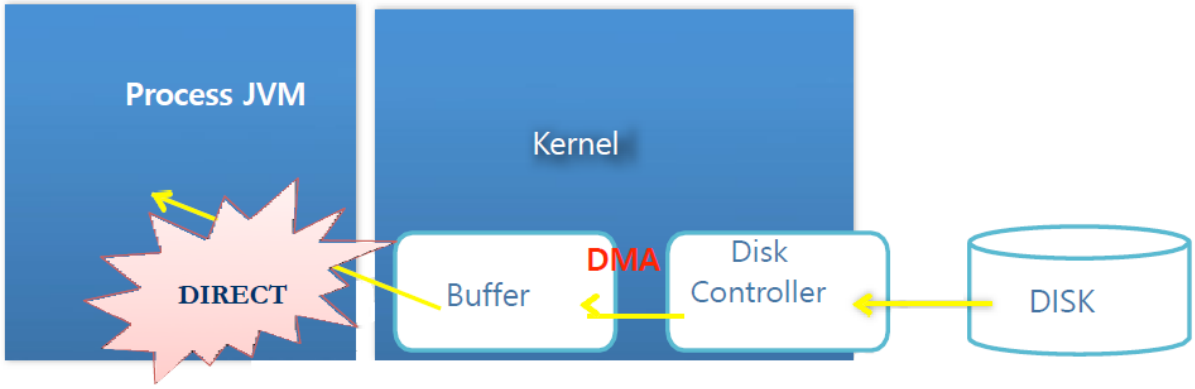
IO	NIO
Stream-oriented	Buffer-oriented
I/O bloccante	I/O non bloccante
	Selectors
	Channels

La caratteristica degli stream di `java.io` è che sono bloccanti, cioè mentre si esegue la lettura o la scrittura il thread viene bloccato per eseguire l'operazione. Una volta terminata, il controllo ritorna al chiamante. Vediamo un esempio:



Tutto questo passaggio è molto lento perché dal buffer del kernel si passa ad un altro buffer nella JVM che è *blocking* (in termini di thread) e si fa una copia. Invece, la caratteristica degli stream di `java.nio` è che **non** sono bloccanti ed hanno un funzionamento anche più chiaro.

La variabile channel indica un canale ottenuto tramite `getChannel()`. Un canale rappresenta una connessione ad un dispositivo I/O fisico (file, socket remoto, un altro programma...).



A differenza di prima è molto più efficiente perché non è bloccante ed il buffer viene usato direttamente dalla JVM senza doverne fare una copia interna. Si salta un passaggio ma che è fondamentale e che legge meno dalla memoria. In particolare

- Non servono thread per bloccare lettura/scrittura su canali separati, tutto molto efficiente
- La read legge solo i dati disponibili
- La write scrive solo i dati che possono essere scritto

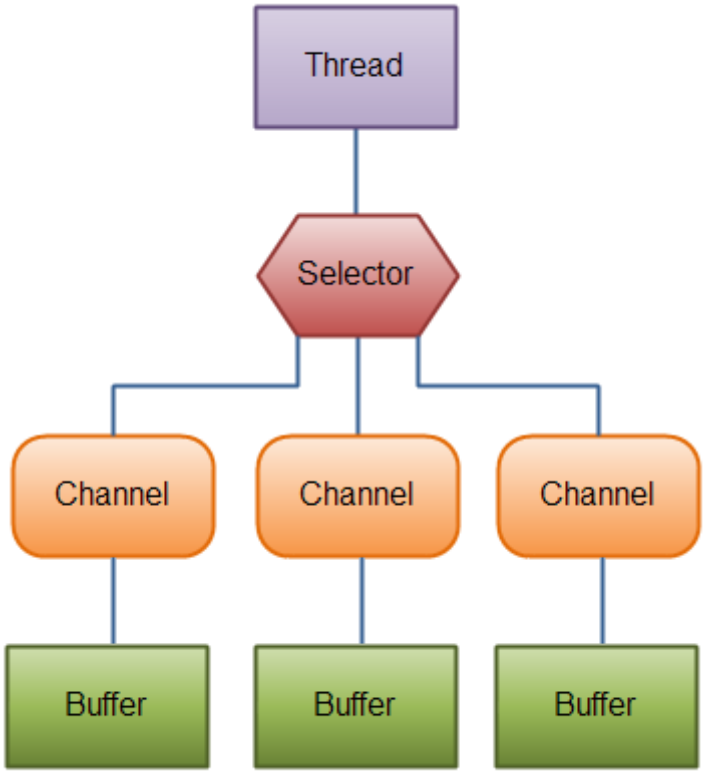
I dati in lettura vanno dallo stream al buffer e i dati in scrittura vanno dal buffer allo stream. Si cerca di usare la memoria fisica del SO per fare operazioni di IO native (veloci) e si cerca di non fare copie inutili. Non-bloccante invio e ricezione.

Tutto l'I/O in NIO inizia con un *channel*; grazie ad esso posso leggere dati da un buffer oppure posso scrivere dati in un buffer. Ci sono diversi canali e buffer che posso utilizzare, ecco i principali:

- `FileChannel`
- `DatagramChannel`
- `SocketChannel`
- `ServerSocketChannel`

- `ByteBuffer`
- `CharBuffer`
- `DoubleBuffer`
- `IntBuffer`

La terza componente è il *selector*; esso permette ad un singolo thread di manipolare più canali. È molto utile se l'applicazione ha più canali parti però a patto che questi abbiano poco traffico.



Il Thread rappresenta il thread principale del nostro programma, poi il selettore collega i vari canali al thread e a sua volta i canali sono collegati ai buffer dai quali pescano i dati e/o scrivono i dati. Per usare un selettore bisogna registrare il canale e poi si chiama il metodo `select()`. Questo bloccherà fino a che non ci sarà un evento pronto per uno dei canali registrati.

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {

  System.out.println("Read " + bytesRead);
  buf.flip();

  while(buf.hasRemaining()){
      System.out.print((char) buf.get());
  }

  buf.clear();
  bytesRead = inChannel.read(buf);
}

aFile.close()
```

In questo esempio vediamo che viene creato prima un canale (`getChannel()`) relativo ad una risorsa (`aFile`), poi si crea un buffer (`ByteBuffer.allocate(48);`) e tale buffer viene collegato al canale (`inChannel.read(buf);`) per poterne fare la lettura. Alla fine il file viene chiuso.

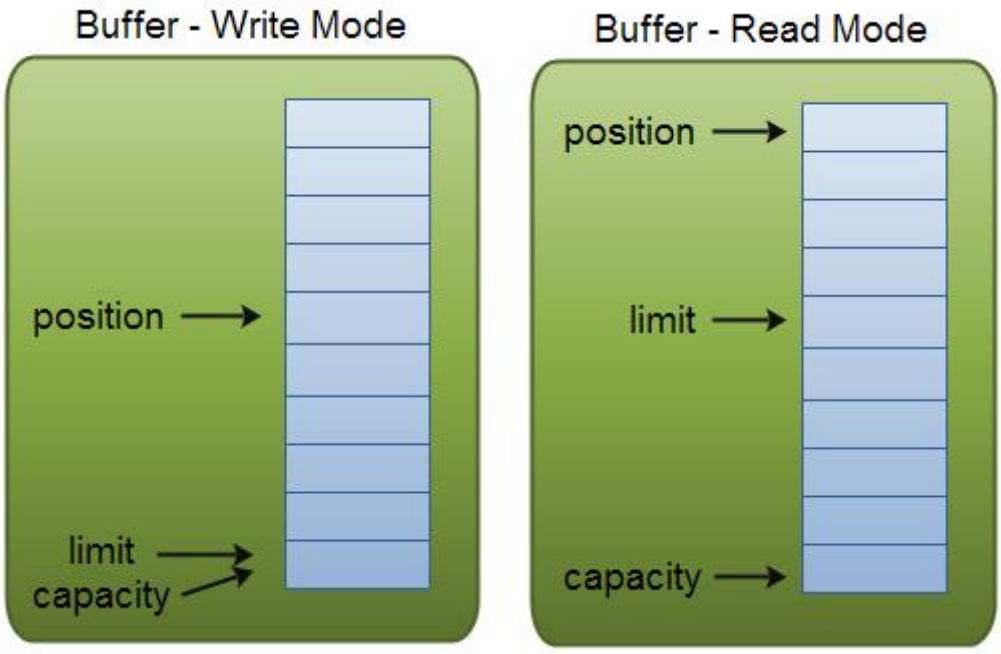
Probabilmente la classe di buffer più importante è la `ByteBuffer`; rappresenta un vettore di dimensione fissata di byte. Si applicano i soliti metodi di get e put. Solitamente quando si usa un buffer si fanno questi quattro passaggi:

1. Scrivo i dati nel buffer
2. Chiamo la `flip()` del buffer (fa in modo che possa leggere i dati dopo la scrittura)

3. Leggo i dati dal buffer
4. Chiudo il buffer

Il metodo `flip()` fa passare un buffer dalla modalità scrittura alla modalità lettura (vedi codice sopra). Abbiamo capito che un buffer è in pratica un blocco di memoria nel quale posso leggere o scrivere in qualsiasi momento; ha 3 proprietà fondamentali:

- *Capaticy*: la dimensione fissata del buffer che NON può cambiare
- *Position*: quando scrive i dati, si parte dalla posizione 0 e poi si tiene traccia di fin dove si è arrivati (max = capacity – 1). Quando leggo i dati lo faccio a partire da una certa posizione; la `flip()` mi riporta a zero per la lettura.
- *Limit*: In scrittura, è il limite di quanti dati posso scrivere e corrisponde con capacity. In lettura invece mi indica quanti dati posso leggere. Più precisamente, in lettura il limit si ferma nel punto in cui ho scritto l'ultimo byte di dato



Con un buffer posso fare diverse operazioni:

- **Creazione**
  Ad esempio con `ByteBuffer buf = ByteBuffer.allocate(48);`

- **Riempimento**
  Ad esempio chiamando il metodo put come `buf.put()`

- **Flipping**
  Fa passare il buffer dalla modalità scrittura a quella di lettura e mette a 0 il *position*

- **Mark**
  Si può fare il `reset()` che riporta a 0 il *position*. Il `mark(x)` segna il punto in cui si vuole che il reset ritorni. Ad esempio, chiamando `mark(3)` segno che `reset()` mi riporterà il *position* a 3 e non a 0.

- **Duplicazione**

Posso ovviamente usare il `compareTo()` e l'`equals()` sui buffer. L'`equals()` è soddisfatto se i buffer sono dello stesso tipo, hanno lo stesso quantitativo di byte restanti da scrivere e i bytes restanti sono di tipo uguale. Il `compareTo()` considera uno più "piccolo" dell'altro basandosi sull'ordine lessicografico.

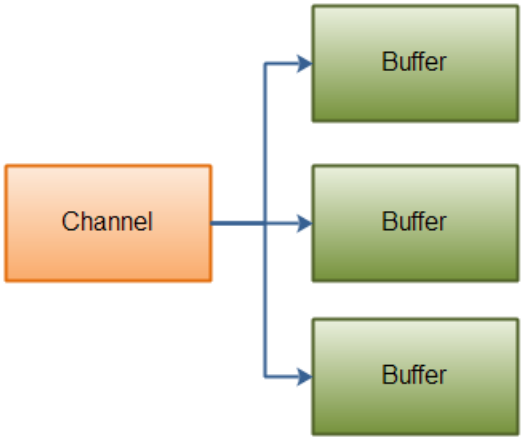
Come abbiamo già visto, la creazione avviene attraverso dei *factory methods* come `allocate(int capacity)`.

Per quanto riguarda i *channels*, solitamente essi vengono associati associati ad oggetti di tipo `RandomAccessFile`, `FileInputStram` o `Socket`. Tutti i canali che implementano `ByteChannel` hanno una `read` e una `write`:

- La `read()` cerca di leggere da un canale quanti più byte possibili ci sono restanti e ritorna quanti ne ha letto (-1 se arriva ad EOF)
- La `write()` cerca di leggere nel canale quanti più byte possibili ci sono nel buffer di origine e ritorna quanti ne ha letti

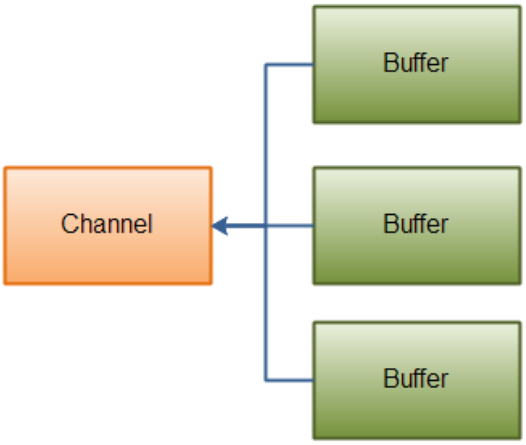
Notare che NIO fornisce supporto built-in a scatter / gather per quanto riguarda la lettura / scrittura nei canali. Più in particolare:

Scattering reads



Legge i dati da un singolo canale e li mette in più buffer distinti

Gathering Writes



Scrive i dati da più buffer in un singolo canale.

Un *selector* permette di esaminare uno o più canali per decidere quali sono pronti per lettura e/o scrittura; in questo modo un singolo thread più gestire più canali. Creo un selector così:

```
Selector s = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(s, SelectionKey.OP_READ);
```

Il canale **deve** essere in non-blocking more per essere usato con il selector quindi non si può usare ad esempio un `FileChannel` perché esso non può andare in non-blocking mode. Vediamo poi che nella register devo anche specificare il "set" di operazioni che mi interessa ascoltare nel channel:

- OP_CONNECT: un canale che si è connesso con successo
- OP_ACCEPT: Un server socket channel che accetta connessioni in entrata
- OP_READ: un canale che ha dati pronti da essere letti
- OP_WRITE: un canale che ha dati pronti da essere scritti

Dopo aver registrato uno o più canali con il selector si può chiamare il metodo `select()` che ritorna i canali pronti per gli eventi ai quali siamo interessati. Notare che:

- La `select()` blocca fino a che almeno 1 canale è pronto per l'evento
- La `selectNow()` non blocca e ritorna subito con qualsiasi canale sia pronto

La java.nio.2 introduce un sacco di features come `FileSystem` factory, la classe `Path` e altre migliorie.

Notare che tutto pare da `FileSystem` e poi si espande:

```
FileSystem fs =
FileSystems.getDefault();
Path p =
fs.getPath("C:\\qualcosa\\a.txt")

try {
    Files.deleteIfExists(p);
} catch (IOException e) {
    //...
}
```

Con Path posso fare delle operazioni di base sui file presenti nel disco come:

- Copiare e spostare
- Controllare gli attributi del file
- Sovrascrivere il file specificato
- Lo spostamento del file può essere un'operazione atomica

Sempre con path ho la possibilità di supporto ai link simbolici, visitare la gerarchia del filesystem (file e cartelle) e gestire i permessi ai file. Notare che:

- Prima di Java SE 1.4: tutto era bloccante e l'uso dei thread era inevitabile
- Java SE 1.4: porta NIO con il non-blocking IO → introduzione di canali e selettori. Tuttavia il non-blocking non era asincrono
- Java SE 7: aggiunta di asincronia all'IO in NIO dentro a `java.nio.channels`

Aggiunta fondamentale quest'ultima perché essendo non bloccante aumenta la scalabilità e si può combinare con l'utilizzo delle `Futures` di java o con l'utulizzo di `CompletionHandler`.

Una **lambda espressione** è una funzione senza nome (sarebbero le anonymous functions di C# e Delphi) che esiste in Java e si implementa usando il nuovo *arrow operator*. Vediamo un esempio che mostra una lambda e una sua implementazione equivalente:

Lambda espressione	Metodo normale
`() -> 123.5`	`double qualcosa() { return 123.5; }`

Una lambda quindi si compone di un'intestazione (le parentesi, dentro alle quali vanno i parametri da dare in ingresso alla funzione), l'*arrow operator* (la freccetta) e il corpo della funzione. Nell'esempio sopra la funzione di destra è stata chiamata `qualcosa` solo per far capire l'equivalenza ma una lambda non ha nome. Vediamo altri due esempi di lambda:

```
//Lambda con corpo singolo
(n, s) -> n * 100 + s

//Equivalente a sopra
(n, s) -> {
    return n * 100 + s
}
```

```
//Lambda con corpo a più righe
(int n, double s) -> {
    int m = s * 3;
    return m * s;
}
```

Da notare che una lambda può avere il corpo fatto da più righe; quando ne ho una sola posso omettere il return e le parentesi graffe scrivendo subito l'espressione col valore da ritornare.

- Nel primo esempio ho una lambda che prende in ingresso due parametri e ritorna un risultato facendo delle operazioni. `n` ed `s` **non** hanno specificato il tipo perché viene dedotto dal compilatore in base ai dati che passiamo in input (fra poco un esempio)
- Nel secondo esempio ho una lambda nella quale specifico il tipo dei parametri

Solitamente è consigliabile specificare il tipo dei parametri ma si può anche non farlo. Ora vedremo il perché attraverso un esempio.

Prima di fare ciò bisogna ricordare che una ***functional interface*** è una interfaccia che contiene un solo metodo astratto (può contenere metodi di default) e solitamente è buona norma marcare tale interfaccia con l'apposita annotazione.

```
@FunctionalInterface
interface Test {
     boolean check(int a, int b);
 }
```

```
@FunctionalInterface
interface Test {
    default int a() { return 2; };
    boolean check(int a, int b);
}
```

Queste sono due interfacce funzionali perché contengono un solo metodo astratto (`check`); un metodo di default ha corpo e non è astratto. Se avessi invece due metodi astratti, allora non avrei più un'interfaccia funzionale. Vediamo un esempio di programma:

```
@FunctionalInterface
interface Test {
    boolean check(int a, int b);
}

public class Main {
    public static void main(String[] args) {
         Test t = (a, b) -> {
             int c = a + b;
             return c % 3 == 0;
         };
         System.out.println(t.check(49, 7));
    }
}
```

Le interfacce funzionali servono per poterci "attaccare" le lambda espressioni, ovvero si può dire che un'interfaccia funzionale costituisce il tipo della lambda.

Come faccio `int a = 5` (5 è un numero intero e ha tipo `int`), allo stesso modo devo fare per le lambda `IntFunz. i = () -> {...}` (a patto che ci sia una compatibilità fra tipo ed espressione)

Vediamo nell'esempio che all'interno del main *"dichiaro il tipo della lambda"* (`Test t`) mettendo alla destra dopo l'uguale una lambda espressione compatibile con la segnatura di `check`. Dato che la segnatura è la seguente

```
boolean check(int a, int b);
```

la mia lambda espressione ti tipo `Test` dovrà ritornare un booleano ed accettare due parametri interi. Questo è il motivo per il quale un'interfaccia funzionale deve avere **SOLO UN** metodo astratto; quando ci assegno una lambda, avendo un metodo solo, il compilatore sa quale scegliere ed esso mi indicherà come strutturare la lambda. Infatti nel testo ho scritto

```
Test t = (a, b) -> {
    int c = a + b;
    return c % 3 == 0;
};
```

e appunto ho una lambda che accetta due parametri `a`, `b` e ritorna un booleano, proprio come dice `check`. Per chiamare la lambda poi faccio `t.check(49, 7)`, cioè uso il metodo dell'interfaccia funzionale. Altra cosa importante che ora si capisce è la seguente.

Dato che l'interfaccia funzionale definisce un metodo con i tipi dei parametri, ci può essere inferenza di tipo sui parametri attuali delle lambda. In pratica: `check(int a, int b)` mi dice che ho due parametri **int**. Bene, allora so che posso chiamare la lambda `(a, b) -> {}` **senza** specificare il tipo di a e b perché questi due saranno interi. Nel codice sopra, avrei potuto scrivere `(int a, int b) -> {}` e sarebbe stato uguale.

Una lambda torna utile soprattutto quando la passo come parametro di una funzione. Supponiamo di avere le seguenti implementazioni:

```
@FunctionalInterface
interface Test {
    boolean check(int a, int b);
}
```

```
public boolean controllo(Test t) {
    int x = 12;
    int y = 26;
    return t.check(x, y);
}
```

Vediamo che la funzione `controllo` accetta il tipo Test che, siccome è interfaccia funzionale, può essere il tipo di una lambda.

```
public static void main(String[] args) {
    boolean SonoUguali = controllo( (a, b) -> a == b );
    boolean MinoreDiB = controllo( (int a, int b) -> {return a < b;} );

    System.out.println(SonoUguali); //stampa false
    System.out.println(MinoreDiB);  //stampa true
}
```

La potenza delle lambda è che permettono di creare "al volo" delle funzioni che permettono di riusare uno stesso metodo per fare diverse cose. In particolare, in questo caso:

- Prima in `SonoUguali` scrivo una lambda che verifica l'uguaglianza dei due numeri
- Poi in `MinoreDiB` scrivo una lambda con i parametri aventi tipo **esplicito** che sarebbe uguale a non metterlo (perché viene automaticamente dedotto guardando il tipo dei parametri in ingresso del metodo dell'interfaccia)

Riassumendo, una lambda è una funzione senza nome che si crea con l'*arrow operator* e può essere assegnata ad un'interfaccia funzionale. A questo scopo, Java offre delle interfacce funzionali già di suo (evitando che ogni volta noi dobbiamo creare un'interfaccia apposita). Vediamo le più importanti:

- **Consumer<T>**

Implementazione

```
public interface Consumer<T> {
    void accept(T t);
}
```

Esempio

```
Consumer<Integer> c =
x -> System.out.print(x);

c.accept(new Integer(3));
```

Serve per eseguire un metodo che non ritorna nulla. Nell'esempio, viene stampato a schermo "3".

- **Function<T, R>**

Implementazione

```
public interface Function<T,R> {
    R apply(T t);
}
```

Esempio

```
Function<String, Integer> f =
(x) -> x.length();

int len = f.apply("Ciao");
```

Serve per eseguire un metodo che ritorna un tipo specificato da R e prende in input un tipo specificato da T. Nell'esempio, viene stampato a schermo la lunghezza della stringa presa in input da `apply`, cioè nel caso dell'esempio 4.

- **Predicate<T>**

Implementazione

```
public interface Predicate<T> {
    boolean test(T t);
}
```

Esempio

```
Predicate<String> f =
(x) -> x != null && x.isEmpty();

boolean test = f.test("Ciao");
```

Serve per eseguire un metodo che ritorna un booleano che verifica se la condizione è stata verificata o meno

Le lambda espressioni possono anche *catturare* le variabili fuori dallo scope della lambda a patto che esse siano **finali** (marcate `final`) o **effettivamente finali** (non marcate col `final` ma comunque non cambiano di valore). Vediamo un esempio:

```
public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(0, 1, 2, 3);
    int val = 10;
    intSeq.forEach(x -> System.out.println(val + x));
}
```

Notare che l'espressione "cattura" la variabile `val`, cioè la prende dall'esterno e se la porta dentro. Se avessi scritto così:

```
final int val = 10;
```

tutto ok perché la variabile è finale e può essere catturare. Se invece scrivessi

```
int val = 10;
int val = 4;
intSeq.forEach(x -> System.out.println(val + x));
```

otterrei un errore di compilazione perché la variabile `val` **NON** è effettivamente finale (cioè vuol dire che dopo la sua dichiarazione senza il `final`, la ho modificata almeno una volta). Se avessi scritto

```
int val = 10;
intSeq.forEach(x -> System.out.println(val + x));
int val = 4;
```

lo stesso non andrebbe bene perché la variabile non è effettivamente finale; anche se ho cambiato il suo valore *dopo* della lambda comunque è cambiato. Quindi non vale. L'ultima nota delle lambda espressioni da tenere a mente è che quando una funzione si aspetta come parametro un'interfaccia funzionale, posso passare sia una lambda che un riferimento ad un metodo compatibile.

```
//Il corpo della lambda verrà attaccato a questa interfaccia
//funzionale che funge da tipo
@FunctionalInterface
interface StringFunc {
    String func(String n);
}

public class Main {

    //Metodo che prende come parametro l'interfaccia funzionale e una
    //stringa; in pratica posso usare le lambda come parametro di
    //questo metodo
    public static String operazione(StringFunc s, String t) {
        return s.func(t);
    }

    //Metodo che stampa i primi tre caratteri di una stringa in input
    public static String primiTre(String a) {
        return a.substring(0, 3);
    }

    //Faccio i test
    public static void main(String[] args) {
```

```
        //Stampo i primi 3 caratteri di "Roberto" usando un riferimento
        //di metodo, ovvero quando trovo un'interfaccia funzionale
        //passo una funzione che abbia la stessa segnatura
        System.out.print(operazione(Main::primiTre, "Robi"));

        //Stampo i primi 3 caratteri di "Roberto" usando una lambda nel
        //solito modo
        System.out.print(operazione((x) -> x.substring(0, 3), "Robi"));
    }
}
```

Da questo esempio si capisce la potenza delle interfacce funzionali. Avendo il metodo con questa segnatura

```
public static String operazione(StringFunc s, String t) {
        return s.func(t);
    }
```

e sapendo che `StringFunc` è una interfaccia funzionale, ho due scelte:

1. Ci metto una lambda come faccio nel secondo pezzo di codice del main. Ovviamente una lambda compatibile con la segnatura del metodo dell'interfaccia (prende una stringa e ritorna una stringa)
2. Ci passo un riferimento ad un metodo di una classe. Il metodo `primiTre` è compatibile con la segnatura del metodo dell'interfaccia perché ritorna una stringa e prende in input una stringa

Quindi capiamo che le interfacce funzionali sono compatibili sia con le lambda che con i riferimenti ai metodi di una classe. In questo caso `primiTre` è statico quindi lo passo con `Main::primiTre`. Se il metodo `primiTre` fosse stato **non** statico allora avrei dovuto fare così:

```
Main m = new Main();
System.out.println(operazione(m::primiTre, "Robi"));
```

Un impiego delle lambda che ha avuto un impatto grandissimo all'interno di Java è stato quello generato dall'introduzione delle **stream API.** Uno stream è un insieme di oggetti, tipicamente provenienti da una collezione tipo List, che vengono processati prima di essere mostrati in output.

- Contenitore di dati (List, ArrayList...) → creo uno stream (un flusso) → imposto dei filtri → esco

```
List<Impiegato> impiegati = ...

impiegati.stream()
    .filter(e -> e.getStipendio() > 2000)
    .map(e -> e.getName())
    .forEach(System.out::println);
```

Creo una lista di oggetti, una lista di `Impiegato`.

Poi creo uno stream, imposto dei filtri sui dati usando varie funzioni e alla fine mostro l'output

Il metodo `stream()` di una collezione ritorna appunto uno stream sul quale posso applicare dei filtri. Il concetto di stream è assimilabile a quello di una pipeline; è come se una scatola (il contenitore `List<Impiegato>`) tirasse fuori TUTTI i suoi elementi su un nastro trasportatore. Poi, delle leve tolgono dal nastro dei pezzi che non soddisfano certi criteri e alla fine viene costruito un contenitore finale che contiene i pezzi rimasti sul nastro. In particolare, dall'esempio sopra, è importante ricordare che uno stream si compone di:

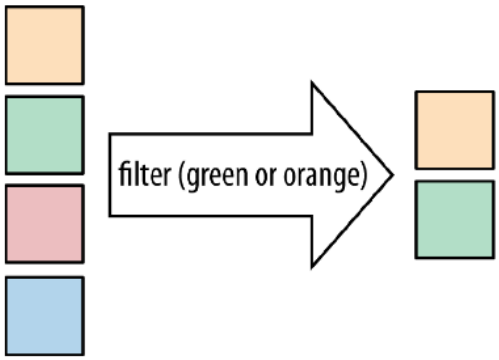
- **Generatore** di stream (sorgente di dati)
- **Zero o più** operazioni intermedie (*lazy*)
- **Una singola** operazione terminale (*eager*)

L'oggetto `impiegati` è il generatore, `map` e `filter` sono operazioni intermedie (ce ne possono essere infinite e `forEach` è un'operazione terminale. Solo quando viene invocata l'operazione terminale la JVM inizia a fare il filtraggio dei dati. Le operazioni intermedie sono dette *lazy* perché non fanno nulla; esse

ritornano un oggetto di tipo Stream (ecco perché si può fare la concatenazione) e dicono soltanto come filtrare i dati. Sarà compito dell'istruzione terminale l'azione di far partire il calcolo.

Vediamo ora una lista di alcune importanti operazioni intermedie che si possono fare su di uno stream con dei relativi esempi d'uso. Tutti i filtri, come vedremo adesso, accettano un'interfaccia funzionale come parametro (vedi sopra) e quindi posso usare le lambda espressioni per specificare in modo conciso le regole

- **filter**
  `Stream<T> filter(Predicate<? Super T> predicate)`



Ritorna uno stream composto da tutti gli elementi che soddisfano la condizione verificata dal predicato.

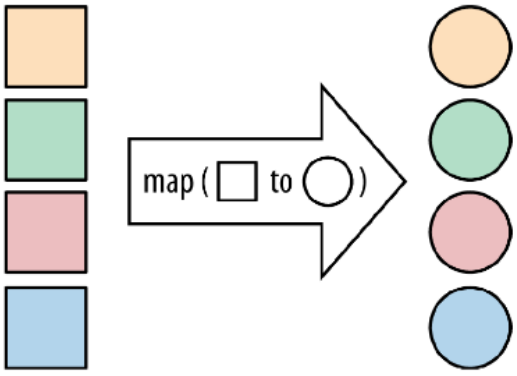
Dall'immagine si vede lo scopo della `filter` che prende solo quadratini del colore specificato.

Una pagina fa dicevamo che un Predicate serve ad eseguire un metodo che ritorna un booleano che verifica se la condizione è stata verificata o meno

```
List<Integer> list = Stream<Integer>.of(1, 2, 3, 4, 5, 6)
    .filter( x -> x.intValue() > 3)
    .collect(Collectors.toList());
```

Questo codice crea una lista usando un metodo di utilità della classe `Stream`. Il filtro estrae tutti i valori per i quali l'oggetto analizzato abbia un valore maggiore di 3. Poi, l'operazione finale `collect` (che vedremo dopo), mostra a schermo il risultato

- **map**
  `Stream<T> map(Function<? Super T, ? extends R> mapper)`



Ritorna uno stream composto da tutti gli elementi composti dallo stream di origine ma con un tipo di dato diverso. Cambia la rappresentazione di dato che si vuole ottenere in output. In altri termini, la `map` fa una conversione da un tipo ad un altro, mantenendo l'ordine.

La `Function<T, R>` ritorna il tipo R ed accetta in input il tipo T

```
List<String> list = Stream<String>.of("a", "bb", "c", "ddd", "e")
    .map( x -> x.length() )
    .filter( x -> x > 1)
    .collect(Collectors.toList());
```

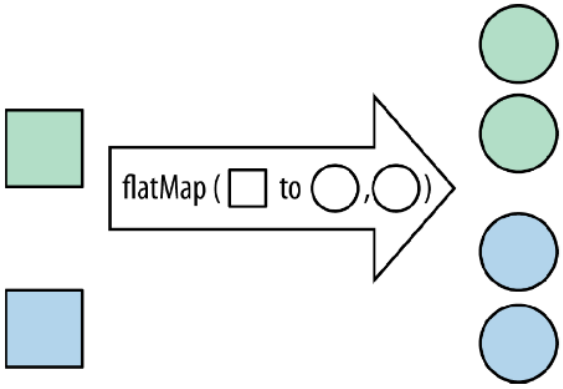
Questo codice crea una lista di stringhe e poi con il metodo `map` "converte" il dato in input (uno `String`) in un intero (la lunghezza della stringa, ottenuta facendo `x.length()`). Poi la filter filtra il risultato mostrando solamente i valori della lunghezza che sono maggiori di uno. Alla fine la variabile list conterrà i valori 2 e 3 (lunghezza di *bb* e lunghezza di *ddd*).

Il `map` quindi si usa quando si vuole lavorare sempre si quei dati ma con una forma diversa; qui ho delle stringhe ma devo lavorare sulla loro lunghezza. Per impostare filtri sulla loro lunghezza, mappo il dato (che è `String`) ad un intero e faccio i miei conti.

```
long list = Stream<String>.of("a", "bb", "c", "ddd", "e")
    .map( x -> x.length() )
    .filter( x -> x > 1)
    .count()
```

In tal caso l'output sarà 2 perché faccio le stesse cose di prima ma il `count()` mi ritorna il numero totale di elementi che ci sono dentro alla pipeline (nello stream).

- **flatMap**
  `<R> Stream<R> flatMap(Function<? Super T, ? extends Stream<? Exteds R>> mapper)`



Produce uno stream di stream; in altre parole fa il processo di linearizzazione di più stream in un flusso unico.

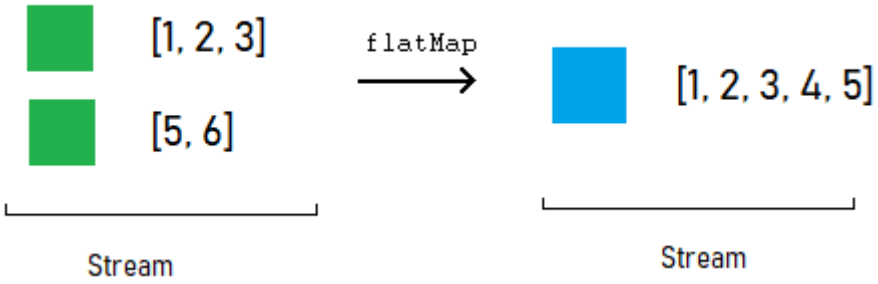
Prende in input uno stream composto da liste di oggetti di un certo tipo (come `Stream<ArrayList<Integer>>`) e ritorna una **unica** lista formata da tutti gli elementi di tutte le liste. Concatena tutti gli elementi delle liste in una lista sola.

```
List<Integer> ls = Stream
        .of(Arrays.asList(1, 2, 3), Arrays.asList(5, 6))
        .flatMap(x -> x.stream())
         .collect(Collectors.toList());
```

La chiamata a `Arrays.asList(1, 2, 3)` ritorna un `List<Integer>` quindi ora lo stream contiene due liste (una è [1, 2, 3] e l'altra è [5, 6]). Un codice equivalente sarebbe questo:

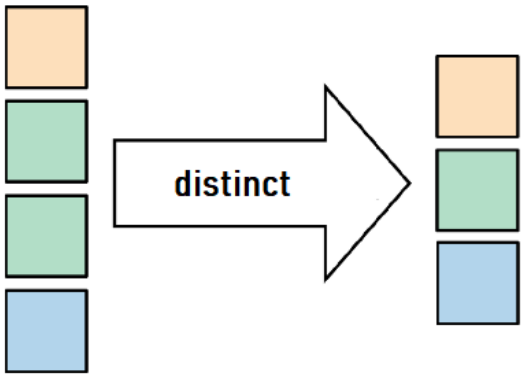
```
var array1 = new ArrayList<Integer>;
array1.add(1); array1.add(2); array1.add(3);

var array2 = new ArrayList<Integer>;
array1.add(5); array1.add(6);

//...
Arrays.of(array1, array2)
//...
```

Ora il comando `flatMap()` fa in modo che da due liste separare, si abbia una sola ed unica lista dalla quale fare varie operazioni intermedie (tipo filtraggio, distinzione o altro).



Prima lo stream era formato da due array (blocchi verdi), poi la flat "appiattisce" tutto, "linearizza" e crea un array unico. Alla fine la `collect()` ritorna una lista percorribile e salva il tutto in `ls`.

- **distinct**
  `Stream<T> distinct()`



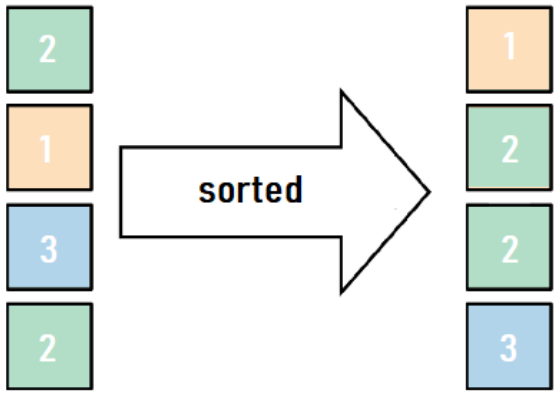
Ritorna uno stream di elementi distinti. Tiene buona solo la prima occorrenza mentre tutte le altre ripetizioni sono ignorate.

Il comando è equivalente logicamente al DISTINCT dell'SQL

```
List<Integer> ls = Stream
        .of(Arrays.asList(1, 3, 2, 3), Arrays.asList(2, 6, 1))
        .flatMap(x -> x.stream())
        .distinct()
        .collect(Collectors.toList());
```

Questo codice fa la stessa cosa di quello sopra ma sta volta c'è il il `distinct()` che fa uscire a schermo i dati senza ripetizioni, cioè [1, 3, 2, 6]. Poi c'è anche la funzione **sorted** che si usa come `distinct()` ma ordina gli elementi secondo il loro ordine naturale; è necessario che gli oggetti implementino l'interfaccia `Comparable`.

- **sorted**
  `Stream<T> sorted()`



Ritorna uno stream di elementi ordinati secondo il loro ordine naturale. Se gli elementi **non** sono `Comparable`, viene lanciata una eccezione.

Ovviamente l'eccezione viene lanciata quando viene applicata un'operazione terminale, perché essa prova ad eseguire il confronto ma, siccome manca l'implementazione dell'interfaccia, scatta l'errore a runtime

```
List<Integer> ls = Stream
        .of(Arrays.asList(1, 3, 2, 3), Arrays.asList(2, 6, 1))
```

```
            .flatMap(x -> x.stream())
            .sorted()
            .collect(Collectors.toList());
```

Questo codice fa la stessa cosa di quello sopra ma sta volta c'è il il `distinct()` che fa uscire a schermo i dati senza ripetizioni, cioè [1, 3, 2, 6]. Da ricordare anche il metodo `peek(Consumer<? super T> action)` che ritorna lo stream di chiamata e in più esegue l'azione specificata in `action`.

Vediamo ora una lista di alcuni operatori **terminali** che servono a far partire il calcolo e stanno sempre alla fine della catena perché devono "chiudere" il processo. Queste mi ritornano **SEMPRE** un risultato **UNICO** a partire da uno stream. Tipo la `count()` mi ritorna un numero (il tot di elementi nello stream).

- `void forEach(Consumer<? Super T> action)`
  Compie un'azione per ogni elemento dello stream
  `> Stream.of( Arrays.asList(1, 2, 3) ).forEach(System.out::print);`

- `long count()`
  Ritorna il numero di elementi presenti nello stream (vedi sopra)

- `boolean allMatch(Predicate<? Super T> predicate)`
  Ritorna true se e solo se tutti gli elementi rendono true il predicato
  `> Stream.of( Arrays.asList(1, 2, 3) ).allMatch( x -> x > -3 );`

- `T reduce(T identity, BinaryOperator<T> accumulate)`
  Il campo *identity* è l'inizializzazione del valore mentre l'altro campo indica come operare su due elementi presi in input. Vediamo un esempio

  ```
  List<Integer> i = Arrays.asList(1, 2, 3);
  Integer sum = i.stream().reduce(0, (a, b) -> a + b + 1);
  ```

  La reduce esegue l'operazione binaria (che prende due valori in input) specificata da una lambda e accumula il valore a progressivo partendo dal valore iniziale specificato.
  In questo caso, sto dicendo di partire a contare da zero e fare la somma di un elemento col suo successivo e di aggiungere 1. Il valore di `sum` sarà 9.

È meglio ricordare ancora che tutte le operazioni intermedie sono **lazy** (non valutate subito) perché vengono eseguite solamente quando viene incontrata un'operazione terminale. Ogni intermedia ritorna un tipo `Stream`. Le lambda che passiamo come parametri **NON** devono fare side effect sullo stream; ad esempio fare una cosa tipo questa:

`qualcheOggetto.stream().filter( x -> qualcheOggetto.add("qualcosa") )`

Non va assolutamente bene e si verrà incontro ad un'eccezione lanciata a runtime. Torna utile sapere che c'è una classe generica in Java che può rappresentare dei valori opzionali. Ad esempio:

```
public Optional<Double> dividi(double a, double b) {
    if (b == 0)
        return Optional.empty();
    else
        return Optional.of( a / b );
}

Optional<Double> divisione = Main.dividi(5, 0);

if(divisione.isPresent())
    System.out.println(divisione.get());
else
```

```
System.out.println("null!");
```

Siccome la divisione (assumiamo di lavorare nei reali) non può avvenire se il denominatore è zero, la funzione `dividi` ritorna un `Optional<Double>`, ovvero un tipo che può essere un `Double` oppure un `null`. La classe `Optional` serve proprio a questo: quando il valore di ritorno è opzionale, conviene usarla perché se non c'è nulla da ritornare, si setta `null`.

- Notare nel codice che se b è zero, ritorno `Optional.empty()` cioè assegno il valore null

Se il valore della divisione esiste, uso il metodo statico `of()` che è un factory di `Optional`. Nel mio codice poi posso facilmente verificare con `isPresent()` se il valore della variabile è null o meno; se è null vuol dire che `divisione` non contiene nulla perché b era zero.

Prima di passare alla distribuzione ed al multi-threading, conviene fare qualche parola sui concetti di DI (**D**ependency **I**njection) e di IoT (**I**nversion **O**f **C**ontrol).

Un buon sistema OO dovrebbe basarsi sull'interazione fra gli oggetti e non fra la dipendenza/parentela fra di essi; si vuole avere un'alta coesione ed un basso accoppiamento. Detta molto in generale, il succo della storia è che conviene evitare l'ereditarietà (che è il tipo più forte di legame) e conviene abbracciare l'uso delle interfacce. Vogliamo che le classi siano enti più "isolati" possibili e che non dipendano da altri pezzi.

"Favor composition over inheritance"

L'ereditarietà comporta il fatto che, in caso di cambiamenti radicali ad una classe che sta in alto nella gerarchia, si può rompere tutta la gerarchia perché le figlie devono adattarsi al cambiamento della madre. Se ci sono molte sottoclassi, potrebbe rompersi tutto. Usare le interfacce è buona cosa perché dicono cosa fare ma lasciano libertà di implementazione. Vediamo un esempio:

```
class ListaFilm {  //classe che elenca una serie di film
      private TrovaFilm f;  //classe con metodi di ricerca in una lista

      public ListaFilm() {
            f = new CSVMovieFinder("movies.csv");
      }
}
```

In questo codice molto semplice sto facendo `new CSVMovieFinder` (sottoclasse di `TrovaFilm`) e ho vari problemi. Intanto, se dovesse cambiare l'implementazione del costruttore dovrei aggiornare TUTTI i costruttori di quella classe, ovunque la abbia dichiarata. Poi, se dovessi pescare i dati no da un CSV ma da un json/SQL dovrei cambiare il tipo di sottoclasse perché la new crea il tipo sbagliato. E se cambiassi `TrovaFilm`? Insomma, la classe ListaFilm **DIPENDE** da `TrovaFilm` tantissimo; un cambiamento a `TrovaFilm` potrebbe rompere tutto.

- Risolvo il problema usando la DI come vedremo adesso.

Un altro modo di approcciare al problema (a parte l'uso di interfacce) è quello della **Dependency Injection**, ovvero una tecnica che mira a minimizzare la dipendenza fra classi. Questa, può essere realizzata principalmente con tre tecniche che vanno usate in base al contesto; usarle ad cazzum non significa che di default sto rispettando i principi. Vediamo i tipi di DI:

1. *Constructor Injection*

```
class ListaFilm {
      private TrovaFilm f;

      public ListaFilm(TrovaFilm film) {
            f = film; //Constructor injection
      }
```

```
        public void qualcosa() { f.stampaListaAvideo(); }
}
```

Per risolvere il problema della dipendenza e "staccare" le due classi, posso "iniettare" la dipendenza via costruttore. In altre parole, faccio la `new` **FUORI E NON DENTRO** alla classe che necessita di usare l'altra classe. Passo via costruttore una reference ad un oggetto che vive fuori.
In questo modo `ListaFilm` non crea più nulla e non ha più l'ownership dell'oggetto; la classe `ListaFilm` ha solo una reference che viene gestita altrove; non interessa altro. Notare quindi che in generale, quando è necessario (come in questo caso) che ci sia dipendenza fra classi (`ListaFilm` deve usare `TrovaFilm`) è meglio usare i principi di dependency injection per ridurre l'accoppiamento tra classi.

2. *Setter Injection*

```
class ListaFilm {
      private TrovaFilm f;

      public ListaFilm() {}

      public void setMovie(TrovaFilm film) {
            f = film;
      }

      public void qualcosa() { f.stampaListaAvideo(); }
}
```

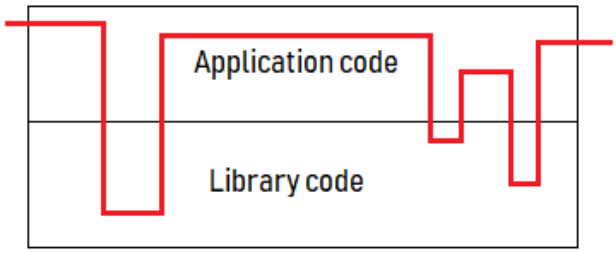
Stessi discorsi di prima solo che qui anziché passare al costruttore la dipendenza, la passo via metodo. Nel costruttore creo l'oggetto ma non ho la necessità di usare `TrovaFilm`, quindi non la passo; piuttosto creo un metodo che mi permette di settare tale variabile.
Diciamo quindi che la Constructor Injection mi indica che ho una dipendenza obbligatoria che va usata per forza, la Setter Injection invece mi dice che si ho una dipendenza ma non è così "forte", posso anche farne a meno.

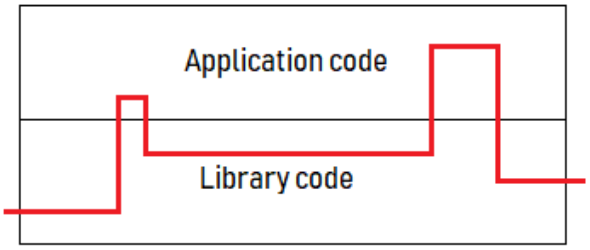
3. *Method Injection*

```
class ListaFilm {
      public ListaFilm() {}

      public void qualcosa(TrovaFilm film) {
            return film.printAll();
      }
}
```

Si tratta del tipo più "soft" di injection perché utilizza temporaneamente un'altra dipendenza (classe o interfaccia); la classe non tiene alcuna reference al suo interno di tale dipendenza. Quindi si fa il minimo indispensabile proprio, toccata e fuga via metodo e basta. Il più basso livello.

Il principio di IoC (detto anche DIP – Dependency Inversion Principle) dice che ogni modulo ad alto o basso livello non deve dipendere da altri moduli ma piuttosto deve dipendere da astrazioni. Si fa inversione di controllo perché si inverte il ruolo di chi gestisce l'applicazione; invece di essere l'app che controlla il flusso di esecuzione, è il framework che controlla il flusso di esecuzione.





L'applicazione ha la maggior parte del controllo del flusso di esecuzione (in rosso); questo è un esempio di ciò che normalmente accade.

Un'applicazione usa una libreria qualsiasi (libreria numerica, di database...) solo quando ne ha di bisogno ma poi è l'applicazione (scritta dal programmatore) che gestisce. Quindi è il programmatore che tramite il codice dell'applicazione controlla il flusso

In caso di inversione di controllo, c'è un framework che controlla l'intero flusso di una applicazione. Infatti si vede che il flusso appartiene molto alla libreria e poco all'app.

Questo accade con le app android ad esempio. Il framework chiama, quando ne ha di bisogno il codice scritto dal programmatore; prima invece era il programmatore che tramite il programma chiamava le funzioni quando voleva lui!

Meglio fare un esempio. Una applicazione Android funziona nell'ambiente Android; quando premo sull'icona di una app c'è il framework (sistema operativo Android) che fa una serie di chiamate a metodi, fa delle new e altre cose per far partire l'app. Il ruolo del programmatore è quello di scrivere una Activity android, cioè il punto di ingresso dell'app (Activity android = il main del C++ per capirci).

Quindi quando l'app parte, il controllo del flusso lo ha in mano il framework (SO Android) perché è lui che chiama i metodi di inizializzazione, gestisce la memoria e gestisce la vita dell'app. Non c'è l'utente che manualmente clicca sull'app, la fa partire invocando i metodi e scegliendo cosa fare. Se ne occupa tutto il sistema operativo.

La dependency injection è una delle tecniche con le quali si può fare inversione di controllo; essa prende il controllo di tutte le azioni base (creazione di oggetti e dipendenze). Famosa è la libreria **Spring** di java che usa molto questo fatto dell'inizializzazione automatica e cerca di passare il flusso di controllo alla libreria il più possibile.

JAVA – PROGRAMMAZIONE CONCORRENTE E DISTRIBUITA

Prima di iniziare, torna molto utile capire la differenza che c'è tra lo stile di programmazione concorrente e lo stile di programmazione distribuito. È molto semplice:

- **Concorrente:** è una tecnica di gestione di più processi su una stessa macchina che operano contemporaneamente e condividendo risorse. Nella pratica si può trattare di un programma che lancia più sottoprogrammi (*thread*) indipendenti che svolgono dei ruoli assegnati
- **Distribuita:** è una tecnica di gestione di più processi che stanno su macchine diverse ma che operano in modo cooperativo per svolgere un unico compito.

Quindi nel primo caso ho un computer unico e più processi che vivono in parallelo, nel secondo caso ho più computer (coi loro processi) che si aiutano a vicenda restando in comunicazione. Vedremo prima l'argomento della concorrenza e poi quello della distrubuzione.

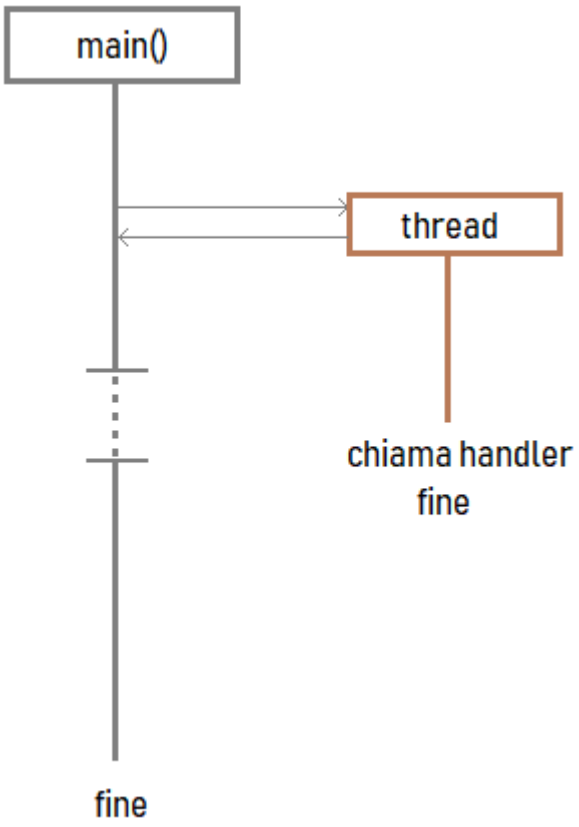
Concorrenza

Abbiamo capito che si tratta di più programmi che corrono in parallelo. Più in dettaglio, si tratta di avere un thread principale (dove vive il main) e poi da esso posso creare degli altri thread secondari che verranno eseguiti concorrentemente. Quindi ci sarà **sempre** un thread principale di origine e poi *n* thread che posso creare da esso.

Il "problema" (che può essere un vantaggio e uno svantaggio) è che questi thread concorrenti che creo vanno per conto loro. Vuol dire che una volta creati, sarà il sistema operativo che li gestisce e noi non abbiamo il potere di controllarli. In generale, si usa questo approccio:

1. Scrivo un programma che abbia un main; quando lo avvio, verrà creato il thread del mail
2. Nel caso dovessi avere operazioni di I/O che possono bloccare (tipo operazioni di rete) faccio partire un thread parallelo che esegue questa operazione onerosa
3. Una volta che il thread parallelo ha finito questo I/O, me lo comunica. Come? Ci sono vari modi, uno dei più comuni è quello di passare ad esempio una lambda che fungerà da "*completion handler*".

Vediamo ora questo esempio. Non si fa sempre così, ci sono tanti metodi ma questo è uno schema molto ricorrente che conviene fissare in testa:



Il main ad un certo punto della sua esecuzione crea un thread (freccia a destra) e dopo che lo ha creato ritorna immediatamente (freccia a sinistra) e continua a fare le sue robe. Durante la creazione del thread, gli passa una lambda (per esempio) che contiene l'azione che il thread deve fare quando ha finito il lavoro.

Intanto il thread esegue le funzioni che gli sono state assegnate e lavora in parallelo col main. Quando avrà finito, potrà chiamare l'handler che gli abbiamo passato. L'handler può essere una lambda che fa così:

```
() → System.out.print("Finito")
```

Oppure se stessi usando una libreria grafica tipo Swing o JavaFX un handler possibile potrebbe essere

```
() → ShowDialog("fine")
```

che mostra a schermo un popup all'utente per avvisarlo che il thread ha finito la sua esecuzione "in background".

La cosa importante da capire quindi è: il main può creare quanti thread vuole ma una volta lanciati **non** ha controllo su di loro. Può verificare solo il suo stato (sveglio, pausa, attivo...) ma non si può sapere quando il thread avrà finito di lavorare. Per sapere quando ha finito, di solito si usano questi event handler che notificano l'utente (ad esempio stampando su console). I principali fattori da tenere in considerazione quando si programma in modo concorrente sono:

1. Non determinismo → visto fino ad adesso, cioè il non sapere nulla sul thread creato
2. Starvation → il thread non riceve abbastanza risorse per fare il suo lavoro
3. Race condition → i thread vogliono accedere alla stessa risorsa condivisa assieme e le modifiche che fanno su di essa si accavallano (mostrando un risultato finale non come quello desiderato)
4. Deadlock → due thread aspettano la disponibilità di una risorsa che però mai sarà disponibile

La buona notizia è che per essere sicuri al 100% di evitare un deadlock bisogna fare sì che **almeno una** di queste condizioni **non** sia verificata:

- Mutua esclusione
- Attesa circolare
- Rimuovere la "preemption" (il SO decide di interrompere un processo per fare spazio ad un altro con priorità maggiore)
- "Hold and wait"

In java ho la classe `Thread` che mi rappresenta il modo di lavorare con più thread. Ho due modi per crearne: uno veloce (implementando `Runnable`) e l'altro più lungo ma con più opzioni disponibili (derivare da `Thread`).

- **Implementare `Runnable`**

  `Runnable` è un'interfaccia funzionale (quindi compatibile con le lambda) che al suo interno contiene il metodo `void run()`. Dentro a `run` ci andrà tutto il codice che verrà eseguito nel thread parallelo e ovviamente è possibile chiamare funzioni esterne (facendo attenzione alla race condition). Vediamo subito un esempio di thread che aspetta tre secondi e stampa del testo:

```
//1. Istanzio il thread
Thread t = new Thread( () -> {
            try {
                Thread.sleep(2000);
                System.out.println("Roberto");
            } catch (InterruptedException i) {
                System.err.println(i.getMessage());
            }
        });

//2. Inizia
t.start();

System.out.print("Ciao sono")
```

  Prima di tutto creo un'istanza della classe `Thread`, necessaria per far partire qualsiasi thread. Nel costruttore passo una lambda che dirà al thread cosa fare; più precisamente sto attaccando una lambda al metodo `run` di `Runnable` (che è interfaccia funzionale). Dato che `sleep()` lancia un'eccezione checked di tipo `InterruptedException`, devo gestirla. Poi è **fondamentale** chiamare il metodo `start()` per far partire il thread.

  Se si fa partire il programma, su console verrà stampato il messaggio *Ciao sono* e dopo due secondi apparirà a schermo anche la scritta *Roberto*. Quindi alla fine l'output è *Ciao sono Roberto* ma Roberto verrà stampato dopo 2 secondi. Alternativa:

```
Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(2000);
                    System.out.println("Roberto");
                } catch (InterruptedException i) {
                    System.err.println(i.getMessage());
                }
            }
 });
```

Questo codice è identico a quello sopra solo che, al posto di usare una lambda, creo una classe anonima che implementa `Runnable`. Scrivere `new Runnable` NON vuol dire creare una istanza dell'interfaccia (che è impossibile!) ma vuol dire creare una classe (anonima, senza nome) che implementa l'interfaccia `Runnable`.

Anche qui creo un thread con una nuova istanza di `Thread` e ci passo una classe che implementa `Runnable`. Notare che non ho chiamato il metodo `start()` e quindi lasciato così, non parte nulla; bisogna usare `t.start();` come prossimo comando per far partire l'esecuzione concorrente al main.

Per rendere tutto ancora più chiaro, vediamo come scrivere l'esempio sopra in modo esteso senza usare una classe anonima:

```
class Roberto implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(2000);
            System.out.println("Roberto");
        } catch (InterruptedException i) {
            System.err.println(i.getMessage());
        }
    }
 }

//Poi nel main...
Thread t = new Thread(new Roberto());
t.start();

//Oppure
new Thread(new Roberto()).start();
```

Ho fatto sempre la stessa cosa di prima ma in modo più esplicito. Quindi riassumendo: per far partire un thread posso creare una nuova istanza di `Thread` e poi gli passo una lambda o una classe che implementa `Runnable`.

- **Derivare da `Thread`**

Il secondo modo per creare un thread è quello di derivare da Thread e fare override del metodo pubblico `run()` (molto simile a prima).

```
class Roberto extends Thread {

    public Roberto(String nomeThread) {
        super(nomeThread);
    }

    @Override
    public void run() {
        try {
            Thread.sleep(2000);
            System.out.println("Roberto");
        } catch (InterruptedException i) {
            System.err.println(i.getMessage());
        }
    }
}
```

```
public static void main(String[] args) {
        Roberto r = new Roberto("");
        r.start();

        System.out.print("Ciao sono ");
}
```

Sto facendo la stessa cosa di prima e l'ouput è lo stesso: vedo subito la scritta *Ciao sono* e dopo due secondo mi appare anche la scritta *Roberto* quindi alla fine otterrò *Ciao sono Roberto*. Anche qui, uso la classe `Thread` per far partire un nuovo thread; più precisamente, sto usando una sottoclasse di `Thread` (che è sempre `Thread`) e quindi chiamo `start()`.
Notare che è necessario un parametro di tipo `String` che corrisponde al nome del thread.
Ancora, come prima, posso stringere ancora di più il main scrivendo:

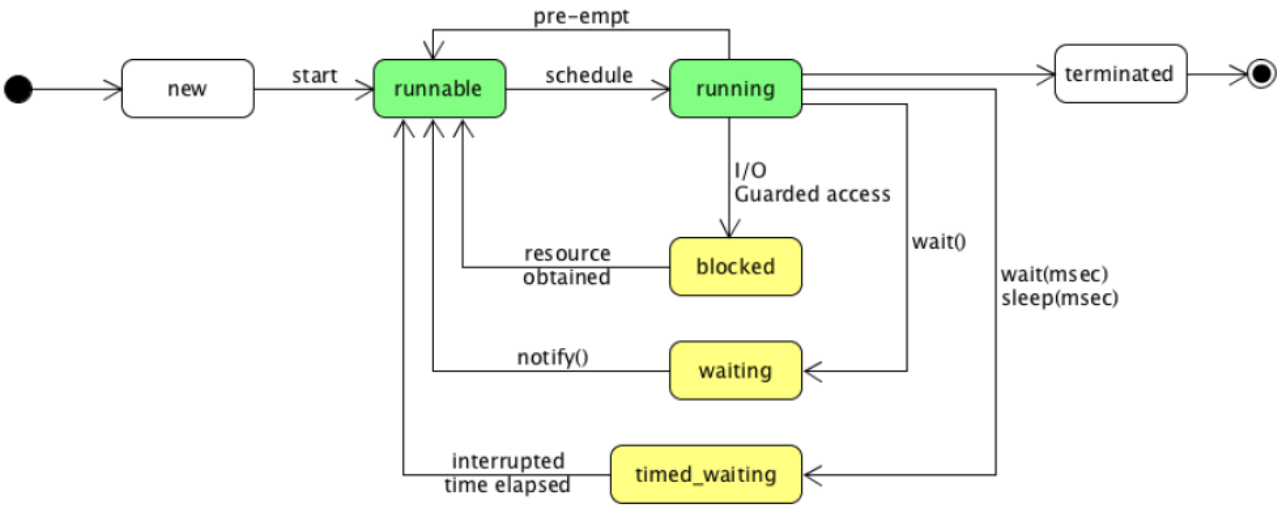
```
new Roberto("").start();
```

Ora sorge spontanea la domanda: quando mi conviene usare un metodo di creazione di thread al posto dell'altro? Ovviamente la risposta è "dipende":

- **Derivare da `Thread`.** La classe `Thread` mi dà molti metodi a disposizione e sono **obbligato** a fare *override* solo del metodo `run()`. Per tutti i discorsi sulla dipendenza fatti, solitamente conviene cercare di evitare l'inheritance. Quindi il consiglio è: derivare da `Thread` solo quando si prevede di fare override di `run()` **E** di altri metodi della classe. Derivare al solo scopo di fare override di `run()` e basta (come nell'esempio) sarebbe uno spreco, tanto vale allora implementare `Runnable` che fa la stessa cosa (implementa run) ma evita l'inheritance!

- **Implementare Runnable.** Praticamente è da preferire quasi sempre questa alternativa perché di un thread 98/100 ci interessa definirne il corpo (= override di `run()`) e il resto dei dettagli non ci importano, vanno bene quelli di default perché si arrangia la JVM a fare ottimizzazione e altre robe.

Quando faccio `new Thread()` posso anche specificare come parametro il nome che potrà essere ottenuto chiamando la `getName()` sull'istanza (in qualsiasi momento). Gli stati di un thread sono:



- All'inizio viene creato e dopo la start, diventa *runnable*. Il thread non è ancora partito, deve aspettare che il SO gli conceda la CPU per eseguire.
- Quando gli viene data la CPU va in stato *running* per un certo tempo e poi torna in "attesa" che sia nuovamente il suo turno per riprendersi la CPU.

- Quando finisce, va in *terminated* e poi il garbage collector si occuperà di cancellare tutto.
- Da *running* può anche passare in altri stati: *blocked* (thread fermo perché aspetta altre operazioni, quindi è bloccato), *waiting* (quando gli viene esplicitamente detto di fermarsi e "dormire" fino a che non sarà risvegliato) e la *timed waiting* (il thread come prima dorme ma a differenza di prima non aspetta segnali per risvegliarsi: aspetta un tot di tempo e poi riparte)

Bisogna ricordare che un thread, essendo alla fine composto da un metodo che va eseguto, può lanciare delle eccezioni. Queste possono essere gestite dentro al thread coi soliti metodi di try-catch. Tuttavia può succedere che un'eccezione non venga gestita. In questo caso, si usa questo metodo:

```
void setUncaughtExceptionHandler( Thread.UncaughtExceptionHandler eh )
```

Questo metodo prende in input una lambda (perché il tipo di `eh` è "attaccabile" alle lambda) che specifica cosa fare in caso di eccezione; in altre parole, questo metodo serve ad impostare un *exception handler* che dice cosa fare. Vediamo un esempio:

```
class Roberto extends Thread { ... }

//poi nel main...
Roberto r = new Roberto("");
    r.setUncaughtExceptionHandler( (Thread t, Throwable e) -> {
        System.out.println("Eccezione gestita qui");
    });

r.start();
```

Se dopo la chiamata ad `r.start();` ci dovessero essere delle eccezioni **NON** gestite da delle try-catch, nella console apparirebbe il messaggio *Eccezione gestita qui* perchè l'exception handler ha una lambda che dice di fare così.

Se dovessimo avere a che fare con tanti thread, fare la new Thread (per esempio) 40 volte diventerebbe costoso in termini di efficienza. Si potrebbe perdere il guadagno che porta il parallelismo, senza contare che poi ci sarebbero 40 istanze da gestire (errori, handler vari etc..). Per questi motivi esistono gli **Executors**, ovvero dei gestori di thread che si affidano al sistema operativo per ottenere efficienza e semplificare il controllo al programmatore. L'interfaccia `ExecutorService` estende l'interfaccia `Executor` la quale contiene il metodo

```
void Execute(Runnable thread)
```

che serve ad aggiungere un nuovo thread al contenitore. Per istanziare un esecutore si usa un metodo factory di `Executors`. Vediamo un esempio:

```
class Roberto extends Thread {

    public Roberto(String nomeThread) {
        super(nomeThread);
    }

    @Override
    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println(getName());
        } catch (InterruptedException i) {
            System.err.println(i.getMessage());
        }
    }
}

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

```
        ExecutorService es = Executors.newFixedThreadPool(2);

        es.execute(new Roberto("uno"));
        es.execute(new Roberto("due"));
        es.execute(new Roberto("tre"));

        Thread.sleep(5000);
        es.shutdown(); //Fondamentale
    }
```

Ho la solita classe da esempio che durante l'esecuzione semplicemente aspetta 1 secondo e poi stampa il nome del thread. Ricordare che `getName()` è un metodo di `Thread` e quindi lo eredito in `Roberto`.

Nel main creo il contenitore di Thread, ovvero `ExecutorService`, facendo uso di un metodo statico della classe `Executors`. Ci sono diversi metodi che posso usare:

- `Executors.newFixedThreadPool`: definisco un insieme fissato di thread che possono eseguire in parallelo (allo stesso tempo). Avendo specificato 2, vuol dire che al massimo 2 thread possono eseguire allo stesso tempo.

  Nell'esempio si vede che, nonostante ci sia il 2 in input, faccio la execute di 3 thread; va benissimo! Il 2 mi sta ad indicare che ci possono essere quanti thread voglio nel contenitore ma fra questi al massimo 2 potranno eseguire in parallelo. Se avessi messo 3, allora tutti e 3 avrebbero eseguito in parallelo

- `Executors.newCachedThreadPool`: crea i thread che gli passo via `execute` ma lo fa solo se strettamente necessario; se si può, l'esecutore cerca di riutilizzare thread già inizializzati

- `Executors.newScheduleddThreadPool`: crea i thread e li fa partire con un preciso schema temporale che specifico

- `Executors.newSingleddThreadPool`: utilizza un solo thread per tutti i compiti

All'interno del main dell'esempio, creo un *Thread pool* (= un gruppo di thread) di dimensione 2 e assegno tre thread. Ho chiamato `Thread.sleep(5000)` per essere sicuro che tutti e 3 i thread finiscano l'esecuzione ma questo approccio è sbagliato (va bene per fare l'esempio, però in realtà si usa una classe `CountDownLatch` e si chiama `await()` per aspettare – però il prof non ha fatto sta cosa quindi ciao).

Alla fine chiamo `es.shutdown()` che è **FONDAMENTALE** perché se non ci fosse stata questa chiamata, il programma non avrebbe terminato perché l'esecutore `es` sarebbe rimasto attivo. Un esecutore rimane sempre in attesa di nuovi compiti da eseguire e tiene sempre la JVM "occupata" quindi bisogna chiamare sempre `shutdown()` per garantire l'uscita.

Fino ad ora abbiamo visto thread che eseguono solo codice ma non ritornano alcun valore; infatti la segnatura del metodo `run()` è

```
public void run() { ... }
```

quindi non ritorna nulla. Esiste un'importante categoria che si chiama **Callable** che rappresenta un thread in grado di ritornare un risultato; come se avessi una funzione multithread. Ha la seguente segnatura

```
interface Callable<V> {

    V call() throws Exception;

}
```

dove dentro a `call()` metto il codice che andrà eseguito in un nuovo thread. Vediamo un esempio che fa uso dell'interfaccia per eseguire una funzione in un thread parallelo:

```
class Test implements Callable<Integer> {
    public Integer call()  {
        Random r = new Random();
        try {
            //Dorme per 1, 2, 3 o 4 secondi
            Thread.sleep((r.nextInt(3) + 1) * 1000);
        } catch (InterruptedException e) {
            System.err.println(e.getMessage());
        }
        return r.nextInt(10);
    }
}

public static void main(String[] args) {
    Test t = new Test();

    System.out.println(t.call());
    System.out.println("Fine!");
}
```

La classe `Test` implementa `call()` in modo molto semplice; dorme per qualche secondo e poi ritorna un numero casuale tra 1 e 10. Importante è vedere cosa stampa il main. La chiamata `t.call()` è **bloccante** ed è giusto/naturale che sia così.

- La chiamata a `call()` deve ritornare un risultato e crea un nuovo thread che eseguirà in parallelo. Non potendo prevedere quando questo thread finirà la computazione e quando mi ritornerà un risultato (per i discorsi fatti prima) **bisogna** far si che `call()` sia bloccante. L'output sarà sempre un numero casuale e poi la scritta *Fine*.

Un'altra alternativa di scrivere il main potrebbe essere la seguente:

```
public static void main(String[] args) {
    Test t = new Test();

    try {
        Future<Integer> f = new FutureTask<Integer>( new Test() );
        System.out.println(f.get());
        System.out.println("Fine!");
    catch (Exception e) {
        System.err.println(e.getMessage());
    }
}
```

Una **Future** è un'interfaccia generica che rappresenta un valore che sarà ritornato da un oggetto di tipo **Callable**. Il metodo `get()` dell'interfaccia `Future` è bloccante come il `call()` di `Callable` per gli stessi motivi. `FutureTask` è la classe concreta che implementa `Future<V>`.

Riassumendo: se voglio che un thread faccia qualcosa senza ritornare valori, estendo `Thread` o creo un nuovo `Runnable` (con lambda da Java 8). Se devo fare un thread che mi ritorna un valore ho due modi

- Creo una classe che estende `Callable` ed implemento `call()`. Ottengo il risultato bloccante con `call()`.
- Creo una `Future` che in costruttore prende un `Callable`. Ottengo il risultato bloccante con `get()`.

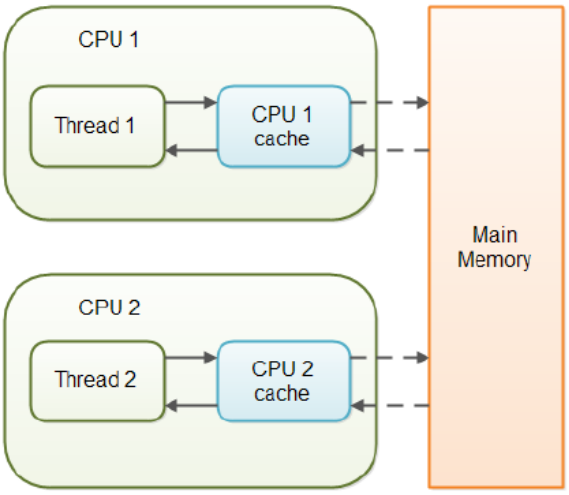
I due metodi sono equivalenti però usare i `Future` ha un vantaggio. Non posso passare ad un esecutore dei *callables* ma posso passare delle *futures*!

```
    ExecutorService es = Executors.newFixedThreadPool(2);
```

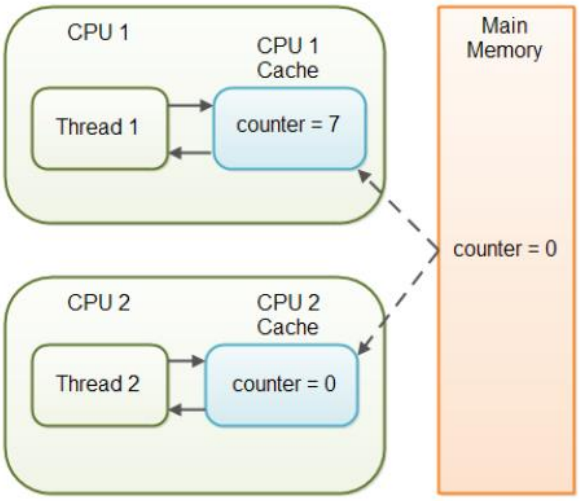
```
Future<Integer> v1 = es.submit(new Test());
Future<Integer> v2 = es.submit(new Test());

try {
    System.out.println(v1.get());
    System.out.println(v2.get());
} catch (Exception e) {
    System.err.println(e.getMessage());
}

es.shutdown();
```

Il funzionamento è molto simile a prima solo che in questo caso uso il `submit(Callable<T> t)` che inserisce la funzione dentro alla scatola che contiene i thread da eseguire in parallelo.

Esiste in Java un tipo particolare di keyword che si chiama `volatile` e lo vedremo velocemente (anche perché non viene usata quasi mai). Una variabile volatile viene sempre letta dalla memoria principale e mai da cache intermedie. Vediamo cosa vuol dire:



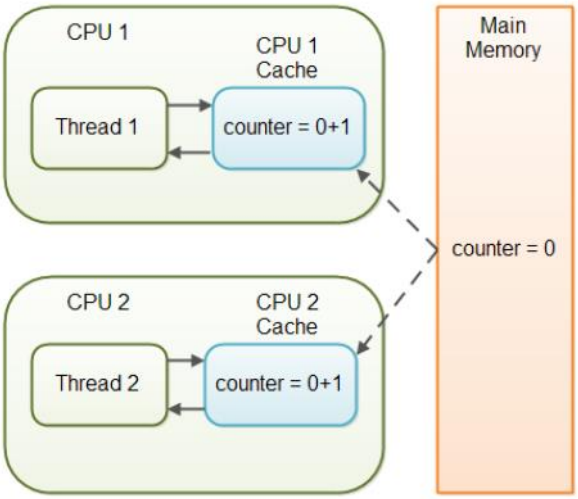
Solitamente le CPU di un computer comunicano con la RAM però prima di accedervi guardano se hanno quello che gli serve salvato nella loro cache personale (perché ci vuole meno ad accedere alla cache che non alla RAM).



Questo significa che thread diversi possono essere smistati in CPU diverse. Ciò accade sempre al giorno d'oggi perché praticamente ogni computer ha più di un core quindi la parallelizzazione può essere sfruttata ancora di più. Thread diversi possono vedere valori differenti. In RAM c'è il valore vero e proprio mentre nella cache un intermedio!

In questo esempio, `counter = 0` ma si vede che la CPU 1 ha `counter = 7` perché deve ancora aggiornare la RAM. Tuttavia CPU 2 ha pescato il valore 0 che è sbagliato perché dovrebbe essere 7 (CPU 1 non ha ancora aggiornato).

Le due CPU non sono in comunicazione e non possono sapere cosa sta nelle cache altrui. Per evitare questi problemi ecco che viene in aiuto la keyword `volatile` che praticamente elimina il caching.



Ora che counter è volatile viene **sempre** letta dalla memoria principale e **mai** dalla cache. Questo fa sì che il dato sia sempre aggiornato ed evita che ci siano inconsistenze fra le due CPU e quindi che escano fuori valori sbagliati nella RAM.

Ho garanzia da parte del compilatore. Tuttavia questa cosa impedisce alcune ottimizzazioni e alcune assunzioni che il compilatore normalmente farebbe. Questa keyword è da usare con attenzione perché è un attimo sbagliare ad usarla e scoprire che proprio questa è la fonte di un bug potrebbe non essere semplice.

Un caso particolare in cui volatile non mi dà la garanzia è quando un thread scrive nella variabile volatile un valore che dipende dal valore che ha appena letto dalla volatile. Cioè, se avessi `volatile int i = 5` e più thread mi fanno `++i` in parallelo, allora non ho più garanzia. Il perché arriva nella riga sotto.

Con tutti questi modelli di multithreading, prima o poi si presenterà il seguente problema (semplificato). Ho una risorsa condivisa *r*, ad esempio un intero, e 10 thread ci accedono in contemporanea per incrementare o decrementare il suo valore. Ci saranno sicuramente dei problemi di **race condition**.

- ***Nota***. Il sistema operativo decide quando fare "context switch", cioè quando dare la CPU ad un thread (= il SO decide quando far funzionare un thread). Tuttavia la CPU concede piccoli intervalli di tempo di esecuzione perché deve simulare il parallelismo. La somma di un numero richiede vari step (estrazione del valore dalla memoria, salvataggio nel registro, incremento e poi salvataggio nella memoria aggiornando il valore).

  Supponiamo che *r* = 1. Potrebbe accadere che mentre un thread A sta facendo la somma, al passo 2 (per esempio) venga interrotto perché la CPU dà spazio ad un altro thread B. Il thread A è *congelato* con *r = 1* ma nel frattempo B può fare la somma col valore che vede e anche lui si salva *r = 1*. Poi A riprende i suoi step e finisce salvando *r = 2*. Anche B però aveva 1 in memoria e salva anche lui il 2.

  Al posto di avere *r = 3* ho *r = 2*. Questo è accaduto perché due thread hanno avuto accesso ad *r* assieme, il SO ha schedulato secondo i suoi piani l'esecuzione e gli step per fare la somma sono stati interrotti.

Per risolvere tutte queste rogne ci sono diverse opzioni che ci fanno un po' perdere in prestazioni però garantiscono una consistenza dei dati.

- **Variabili atomiche**

  ```
  int i = 3                                   //non thread safe
  AtomicInteger i = new AtomicInteger(3);     //thread safe
  ```

  Nel primo caso ho una variabile normale non thread safe che soffre del problema descritto nella nota. Nel secondo caso ho una variabile che non soffre di problemi di concorrenza. Questo accade perché `AtomicInteger` **garantisce** che ogni operazione sia atomica.

  Dire che un'operazione è atomica vuol dire che viene eseguita in blocco come se fosse una cosa sola. Quindi per fare la somma ho sempre i 4 passi descritti prima però sta volta vengono eseguiti tutti di fila e non accade che il SO mi interrompa nel mezzo.

  ```
  AtomicInteger i = new AtomicInteger(3);
  int s = i.incrementAndGet();
  System.out.println(s);
  ```

Le variabili atomiche non hanno operatori ma uso metodi per fare le operazioni (come la somma). Sono sicuro che la somma verrà eseguita come pezzo unico quindi non avrò più i problemi di prima. Ovviamente in ambienti a singolo thread non conviene usare `AtomicInteger` perché avrei un overhead inutile.

- **Metodi synchronized**

  La stessa cosa vista per la somma può accadere ad un metodo. Se `void Roberto()` impiega 500ms ad eseguire, a causa del context switch del SO un altro thread potrebbe chiamare la stessa funzione sulla stessa istanza. Avrei così due metodi che eseguono la funzione allo stesso tempo e chissà che casino verrà fuori coi dati.

  Vediamo un esempio semplice che mostra tutto quanto:

```
class Stampa {
    //Stampa una parentesi quadra, il messaggio passato in
    //input e dopo un secondo stampa l'altra parentesi quadra
    public void stampa(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.err.println(e.getMessage());
        }
        System.out.println(" ]");
    }
}

class ChiamaMetodo implements Runnable {
   private Stampa s;

   public ChiamaMetodo(Stampa s) {
       this.s = s;
   }

    @Override
    public void run() {
        s.stampa("Roberto");
    }
}

//nel main...
public static void main(String[] args) {
    Stampa s = new Stampa();

    ChiamaMetodo c1 = new ChiamaMetodo(s);

    new Thread( c1 ).start();
}
```

  Questo codice stampa *[Roberto]* (prima scrive *[Roberto* e dopo un secondo *]*). Fin qui tutto bene, in particolare notare che:

  - La classe `Stampa` semplicemente stampa una parentesi quadra, un messaggio e dopo un secondo stampa un'altra parentesi quadra

- La classe `ChiamaMetodo` implementa Runnable (perché deve correre su thread) e semplicemente prende un parametro di una istanza che poi userà.
- Il main crea `c1` facendolo partire in un thread nuovo. N

Tuttavia se ora all'interno del mai scrivessi questo codice il risultato cambierebbe:

```
Stampa s = new Stampa();
ChiamaMetodo c1 = new ChiamaMetodo(s);
ChiamaMetodo c2 = new ChiamaMetodo(s);

new Thread( c1 ).start();
new Thread( c2 ).start();
```

```
//Output che ho ottenuto
[ Roberto[ Roberto ]
 ]
```

```
//Output che mi sarei aspettato
[ Roberto ]
[ Roberto ]
```

Mi aspetterei che mi stampasse due *[Roberto]* uno sotto all'altro ma invece mi stampa quello che c'è in figura a sinistra.

Questo accade perché sto passando **LA STESSA ISTANZA** (cioè `s`) alla classe che verrà eseguita su thread paralleli; questo vuol dire che dopo le due istruzioni di `start()` ho due thread the stanno accedendo **ASSIEME** allo stesso metodo `stampa()` ! Per evitare questo problema facci così:

```
class Stampa {
    public synchronized void stampa(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.err.println(e.getMessage());
        }
        System.out.println(" ]");
    }
}
```

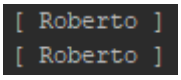
Avendo aggiunto `synchronized` ho fatto in modo che il metodo `stampa` possa essere acceduto soltanto da un thread alla volta; non ci possono essere due thread che accedono (come prima) assieme in concorrenza (Il `var` è come `l'auto` del C++, permette di dedurre il tipo.). L'output è

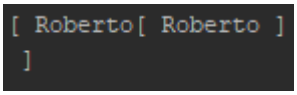
```
[ Roberto ]
[ Roberto ]
```

ed è proprio quello che mi aspettavo. Se non mettessi `synchronized` è come se non mettessi nessun filtro che dice "uno alla volta"; infatti (senza) i thread dopo lo start partono subito e, potendo entrare assieme, eseguono il metodo stampando entrambi *[Roberto*. Per questo ottenevo la stampa strana all'inizio, perché tutti e due accedevano assieme. Avendo messo il blocco ora accedono uno per volta. Importante notare che:

```
Stampa s = new Stampa();

var c1 = new ChiamaMetodo(s);
var c2 = new ChiamaMetodo(s);
```

```
Stampa s = new Stampa();
Stampa ss = new Stampa();

var c1 = new ChiamaMetodo(s);
var c2 = new ChiamaMetodo(ss);
```






```
// Il var è come l'auto del C++, permette di dedurre il tipo.
```

Il codice a sinistra è quello che abbiamo usato fino ad adesso e tutti d'accordo che funzioni. Il codice di destra (nonostante ci sia il `synchronized`) non funziona. Questo succede perché il `synchronized` permette di bloccare l'accesso ad un metodo chiamato **DALLA STESSA ISTANZA** e no da istanze diverse.
A sinistra sto creando i due `ChiamaMetodo` su s, quindi i thread paralleli useranno la stessa istanza (`s`). A destra sto creando i due `ChiamaMetodo` su `s` ed `ss` che sono cose diverse, quindi i thread paralleli useranno istanze diverse.

Detto tutto questo, adesso si capisce perché esistono le *Concurrent data structures*, ovvero le versioni thread safe delle strutture dati come la `HashMap`. Una HashMap non è thread safe perché ci possono essere problemi durante l'inserimento (due thread inseriscono allo stesso tempo) e per questo esiste la sua

controparte thread safe chiamata `ConcurrentHashMap`. Fornisce i metodi soliti ma che sono thread safe e possono correre su diversi thread per velocizzare le operazioni di inserimenti, ricerca o modifica. Importante è che tutte le operazioni che si fanno su strutture concorrenti **NON** devono dipendere da stati condivisi o dall'ordinamento degli elementi.

Un altro esempio è l'interfaccia `BlockingQueue` che aggiunge alla classica Queue i metodi per accodamento e prelievo che siano sicuri in ambiente con più thread.

Accodamento

metodo	risultato negativo
`add(e)`	eccezione
`offer(e)`	`false`
`put(e)`	attesa
`offer(e, time, unit)`	attesa limitata

Prelievo

metodo	risultato negativo
`remove()`	eccezione
`poll()`	`null`
`take()`	attesa
`poll(time, unit)`	attesa limitata

Manca un ultimo caso da vedere riguardo le variabili locali e la condivisione dello stato fra thread. La classe chiamata `ThreadLocalVariable` garantisce che la stessa variabile abbia un valore indipendente e separato per ciascun Thread. In pratica ogni thread che accede a questa variabile ha una sua copia indipendente; vediamo la differenza dalle altre:

```
int i = 2;
ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 2);
```

Viene inizializzata via metodo "(circa) factory" che prende un `Supplier` in input e assegna quel valore alla variabile. Tuttavia, assomigliano molto alle variabili globali quindi sarebbe meglio fare attenzione e cercare di usarle con cautela (oppure proprio mai).

IMPORTANTE. Il succo di tutta la storia è il seguente: quando si ha a che fare con ambienti multithread, cercare di evitare di avere oggetti condivisi. Un oggetto condiviso va bene se acceduto da più thread **SOLO IN LETTURA**; quando più thread invece fanno scrittura non va bene. Un modo facile è quello di usare il `synchronized` e/o le variabili atomiche. La *sezione critica* è quell'area nella quale ci sono dati condivisi da più thread e lo scopo è quello di far sì che non si crei una sezione critica (e per risolvere faccio sì che un thread alla volta sia dentro).

I due metodi descritto sono da preferire quasi sempre perché lasciano al sistema operativo tutta la parte di gestione dei lock e degli accessi garantendo così che tutto vada bene. Si arrangia la JVM a fare tutto e a far sì che uno solo per volta abbia accesso. Tuttavia in alcuni casi serve fare i fenomeni e gestire a mano tutto ciò, quindi senza usare il synchronized. Ci sono diverse tecniche:

- **Wait / Notify**

  Nella classe Object sono definiti `wait()`, `notify()` e `notifyAll()` come metodi `final` e quindi essi sono disponibili a tutte le classi Java.

  Importante notare che questi 3 possono essere usati **SOLO ED ESCLUSIVAMENTE** dentro ad un blocco synchronized. In breve, questi servono a:

  - `wait()`: il thread sul quale viene chiamato va a dormire lasciando il monitor libero (così un altro può entrare). Verrà risvegliato quando qualche altro thread entra nello stesso monitor e chiama la `notify()` o `notifyAll()`.
    In pratica con `wait()` mando a dormire un thread e libero il posto, con `notify()` lo sveglio e quando può si riprende il posto

  - `notify()`: sveglia un thread in sleep che ha chiamato la `wait()` (sullo stesso oggetto)

  - `notifyAll()`: sveglia tutti i thread in sleep che hanno chiamato la `wait()` e ad uno di essi verrà garantito subito l'accesso (poi a seguire tutti gli altri)

  Normalmente `wait()` aspetta un `notify()` (o `notifyAll()`) che risvegli l'oggetto ma in alcuni rari casi è possibile che si verifichi la *spurious wakeup*, ovvero che un thread si svegli da solo senza avere chiamato una notify. Vediamo un breve esempio:

```
public synchronized void prova() {
    while (isValid) {
        try {
            wait();
        } catch (InterruptedException e) {
            System.err.println(e.getMessage());
        }
    }
    notify();
}
```

  Nell'esempio, la variabile `isValid` è da qualche parte impostata e quando essa è `true` allora il chiamante va a dormire. Notare che per chiamare `wait()` è necessario gestire l'eccezione checked di tipo `InterruptedException`. Quando qualche thread cambierà il valore di `isValid` e qualcuno raggiungerà il `notify()`, allora il thread che dormiva verrà svegliato e proseguirà col suo lavoro.

- **Lock**

  Un *monitor* è una zona nella quale ci sta un solo thread. Prima col `synchronized` si arrangiava la JVM ad aprirlo/chiuderlo per bloccare i thread; adesso con il wait/notify devo occuparmene io. Se in più ho bisogno di controllare le condizioni di blocco, allora devo passare all'uso dei Lock.

  Il lock è una alternativa al synchronized però devo fare tutto io a mano e no lasciar fare alla JVM. Prima di accedere ad una risorsa condivisa, il lock che la protegge viene acquisito (`lock()` oppure `tryLock()`); una volta finito di usare quella risorsa viene rilasciato (`unlock()`) il lock. Se un secondo thread cerca di accedere mentre il lock è acquisito, questo verrà sospeso fino a che il lock non viene rilasciato. Questi metodi sono disponibili grazie all'interfaccia `Lock`:

  - `void lock()`: acquisisce il lock proteggendo la risorsa

- `bool tryLock()`: prova ad acquisire il lock e ritorna un booleano che indica il successo/insuccesso però **NON** aspetta se il lock non è disponibile. Al contrario, ritorna true se ha preso il lock, altrimenti false

- `void unlock()`: rilascia il lock

In generale quindi: chiamare `lock()` per ottenere un lock (accesso esclusivo alla risorsa); se non è possibile, allora `lock()` aspetterà. Per rilasciare, usare `unlock()`. Per verificare se un lock è disponibile usare il `tryLock()` che, come visto, non blocca e ritorna semplicemente un vero o falso. A livello pratico, Java fornisce un'implementazione di Lock con la classe `ReentrantLock`, ovvero un lock che può essere acceduto più volte dal thread che attualmente detiene il lock (e non da altri → quello che ha il lock entra ripetutamente quando vuole, ma solo lui).

Ovviamente, nel caso di `ReentrantLock` bisogna che ad *n* lock corrispondano anche *n* unlock. Tutte queste sono difficoltà da affrontare se si vuole gestire manualmente il discorso di accesso sincronizzato ai thread.
Con `Condition` posso gestire, su un solo lock, più condizioni di attesa distinte. Chiamando sull'oggetto lock il metodo `newCondition()` ritorno un oggetto `Condition` associato a quel lock. In particolare, sull'oggetto Condition ritornato posso chiamare

- `void await()`
- `void signal()`
- `void signalAll()`

che hanno lo stesso funzionamento di `wait()`, `notify()` e `notifyAll()`.

Con i lock ad esempio si risolve il problema del produttore/consumatore.

- **Semafori**

I semafori si utilizzano per controllare l'accesso ad un insieme omogeneo di risorse; è simile ad un lock ma tiene un conteggio invece del semplice stato libero / occupato. Una `Semaphore` controlla l'accesso ad una risorsa condivisa usando un contatore: se questo è maggiore di zero, l'accesso è consentito. Se il contatore è zero, l'accesso è negato.

La variabile contatore conta i *permessi* che danno accesso alla risorsa e quindi per accedervi, un thread deve ottenere un permesso dal semaforo. Quindi facendo uno schema:

- Per usare un semaforo, il thread che vuole accesso alla risorsa condivisa prova ad ottenere un permesso.
- Se il contatore del semaforo è maggiore di zero, allora il thread acquisisce il permesso ed il contatore viene diminuito di 1.
- Se il contatore del semaforo è zero, allora il thread sarà bloccato fino a che un permesso potrà essere acquisito (ovvero quando qualcuno che già lo ha finisce il lavoro e lo cede)

Quando un thread finisce, restituisce il per messo al Semaforo (= incrementa di 1 il contatore) e lascia subito spazio ad un altro Il suo costruttore è questo:

```
Semaphore(int num)
Semaphore(int num, boolean how)
```

- `int num`

Specifica il conteggio di permessi iniziale quindi mi dice quanti thread possono accedere alla risorsa in qualsiasi momento. Di default, i thread in attesa ottengono il permesso che si libera **A CASO** (semaforo *non fair*).

- `boolean how`

  Se impostato a `true` si è sicuri che i thread in attesa otterranno il permesso in base all'ordine secondo il quale lo hanno richiesto (semaforo *fair*). Quindi se questo parametro è `false`, quando si libera un posto (= c'è un *permit* libero) un thread a caso becca il posto e può entrare. Se invece `how` è `true` allora quando si libera un posto non entra uno a caso ma, secondo una politica FIFO, entra il primo thread nell'ordine.

Ovviamente un semaforo **non** fair è molto più efficiente di uno fair perché non deve mantenere una coda e sceglie subito a caso uno a cui dare il permesso di entrare. Per ottenere il permesso chiamo uno di questi metodi

```
void acquire() throws InterruptedException;
void acquire(int num) throws InterruptedException;
```

che andrà nel primo caso a diminuire (se maggiore di zero) il contatore del semaforo, nel secondo caso invece il contatore del semaforo verrà diminuito `num` volte. Per rilasciare il permesso si usa

```
void release();
void release(int num);
```

con lo stesso criterio di prima. Notare che il valore di iniziale del semaforo può essere superato e può anche essere non negativo all'inizio (vuol dire che nessuno entra subito, bisogna incrementare un paio di volte). A differenza di un lock, un semaforo può essere rilasciato da un thread diverso da quello che lo ha acquisito.

Il metodo `tryAcquire()` ritorna un booleano che indica se è stato acquisito il permesso o meno (ritorna subito) ma attenzione perché in grado di **violare** la fairness del semaforo (se impostata). Vediamo un esempio:

```
//Classe che contiene la variabile condivisa
class Shared {
    static int count = 0;
}

//Clase che aumenta fino a 5 il valore di count
class IncThread implements Runnable {

    private String name;

    private Semaphore sem;

    public IncThread(Semaphore s, String n) {
        name = n;
        sem = s;
    }

    @Override
    public void run() {
        System.out.println("Starting " + name);

        try {
            //Ottengo un permesso
```

```
            System.out.println(name + " vuole permesso");
            sem.acquire();
            System.out.println(name + " ha il permesso");

            for(int i = 0; i < 5; ++i) {
                //Incrementa di 1 count e poi stampa
                System.out.println(++Shared.count);

                Thread.sleep(100);
            }

        } catch (InterruptedException e) {
            System.err.println(e.getMessage());
        }

        System.out.println(name + " rilascia permesso");
        sem.release();
    }
}

//Classe che decrementa fino a 0 il valore di count
class DecThread implements Runnable {

    private String name;

    private Semaphore sem;

    public DecThread(Semaphore s, String n) {
        name = n;
        sem = s;
    }

    @Override
    public void run() {
        System.out.println("Starting " + name);

        try {
            //Ottengo un permesso
            System.out.println(name + " vuole permesso");
            sem.acquire();
            System.out.println(name + " ha il permesso");

            for(int i = 0; i < 5; ++i) {
                //Decrementa di 1 count e poi stampa
                System.out.println(--Shared.count);

                Thread.sleep(100);
            }

        } catch (InterruptedException e) {
            System.err.println(e.getMessage());
        }

        System.out.println(name + " rilascia permesso");
        sem.release();
    }
}
```

Un po' lungo ma facile. La classe `Shared` ha una variabile statica che sarà condivisa e protetta da un semaforo. La classe `IncThread` si prende in input il semaforo per gestire l'accesso a `count` e una stringa che indica il nome del thread. Nel corpo invece prende un permesso, incrementa di 5 ogni 100 millisecondi e poi rilascia il semaforo. `DecThread` fa la stessa cosa ma decrementa di 5.

```
public static void main(String[] args) {
    //Permesso unico, per fare l'esempio
    Semaphore sem = new Semaphore(1);

    //Faccio partire i thread in parallelo
    new Thread(new IncThread(sem, "A")).start();
    new Thread(new DecThread(sem, "B")).start();
}
```

Qua faccio partire i due thread assieme in concorrenza e l'output sul mio pc è questo ma su altri pc potrebbe essere un po' diverso (diverso nel senso che A e B partono in momenti diversi ma il meccanismo del semaforo va sempre.

```
===== OUTPUT =====

Starting A
Starting B
A vuole permesso
A ha il permesso
B vuole permesso
1
2
3
4
5
A rilascia permesso
B ha il permesso
4
3
2
1
0
B rilascia permesso
```

Vediamo che parte prima A e poi B quindi "vince" A perché ottiene per prima il permesso. B invece "perde" e va in attesa. Infatti si vede che

```
A vuole permesso
A ha il permesso
B vuole permesso
1 ...
```

A ha ottenuto il *permit* del semaforo e sta facendo il suo lavoro (incrementare counter) mentre B aspetta. Una volta che A finisce

```
...5
A rilascia permesso
B ha il permesso
4
```

A lascia il *permit* abbandonando il semaforo, il contatore del semaforo sale ad 1 e B ottiene il permesso.

Il semaforo dell'esempio non è fair perché non è impostato nessun parametro booleano nel costruttore. Se avessimo fatto `new Semaphore(2)` allora l'output sarebbe stato:

```
===== OUTPUT =====

Starting B
Starting A
B vuole permesso
A vuole permesso
A ha il permesso
B ha il permesso
1
0
-1
0
-1
```

Ora il semaforo ha due permessi e quindi due thread possono accedere a counter assieme. In questo caso vediamo che tutti e due ottengono il permesso

```
B vuole permesso
A vuole permesso
A ha il permesso
B ha il permesso
...
```

```
0
0
-1
0
-1
B rilascia permesso
A rilascia permesso
```

E ciascuno fa il suo compito di incrementare e decrementare di uno. Si vede che l'output infatti non è uniforme perché entrambi stanno eseguendo assieme.

```
B rilascia permesso
A rilascia permesso
```

Alla fine entrambi escono. Eseguendo questo codice ogni volta si otterranno sempre risultati diversi.

Stream Paralleli

Abbiamo già visto tutto un discorso lungo sugli stream e abbiamo capito che sono un nastro trasportatore che fa selezione sugli oggetti nel nastro e alla fine mostra solo quelli che soddisfano i criteri. Gli stream possono lavorare su un insieme di dati che sono potenzialmente infiniti. Offrono quindi:

- delle API per comporre passi di elaborazione su una serie di oggetti
- un approccio funzionale al trattamento dei dati
- test in isolamento
- un nuovo stile espressivo con una sintassi che cerca di essere intuitiva
- tanti comandi per fare tante cose sugli oggetti

La cosa importante è che il passaggio da esecuzione in sequenziale ad esecuzione in parallelo non richiede nessuna modifica al codice. Ad esempio

```
//Versione sequenziale
qualcosa.stream().forEach( x -> System.out.println(x); );

//Versione parallela
qualcosa.parallelStream().forEach( x -> System.out.println(x); );
```

Le collezioni si focalizzano sulla performance dell'accesso ai contenuti, uno stream invece si concentra sul calcolo dei contenuti e quindi serve un algoritmo di calcolo definito da noi (con le funzioni intermedie tipo `map()`, `filter()`, `sorted()`...). Non possiamo però assumere che in un `parallelStream()` gli oggetti vengano acceduti in sequenza, quindi l'algoritmo che adotteremo dovrà essere "indipendente" dall'ordine degli oggetti.

Viene lasciata libera scelta sull'ordine di esecuzione al fine di poter ottimizzare il parallelismo. La JVM può prendere decisioni in base a vari fattori come il numero di thread nel computer ad esempio. Inoltre la sorgente di uno Stream può dichiarare alcune *caratteristiche* (dette **stream flag**) che influenzeranno:

- gli operatori intermedi (che possono fare verifiche o controlli sui dati)
- gli operatori terminali (usano tali flag per prendere decisioni sull'esecuzione dello stream; in base ai flag quindi viene deciso il modo migliore per fare i conti)

L'operatore terminale sa quali sono tutte le operazioni intermedie e quindi può prendere delle decisioni su come organizzarsi il lavoro. Valgono ovviamente tutti i discorsi fatti per gli stream seriali riguardo cosa fanno le funzioni tipo `sorted()`, `distinct()` e via così.

Importante ricordare che le operazioni passate ai metodi **NON** devono interferire con gli elementi dello stream.

Splititerator

La JDK 8 ha introdotto l'interfaccia `Splititerator` che consente anche di attraversare una lista sorgente in modo parallelo con più rami di esecuzione indipendenti fra di loro. Per usarlo è molto semplice: basta chiamare il metodo `tryAdvance()` fino a che non ritorna `false`. Vediamo un esempio:

```
ArrayList<Double> array = new ArrayList<>();
array.add(1.0);
array.add(2.0);
array.add(3.0);
array.add(4.0);
array.add(5.0);

System.out.println("Elementi presenti: ");

Spliterator<Double> s = array.spliterator();
while (s.tryAdvance( n -> System.out.println(n) ));
```

Vediamo che la chiamata ad `array.spliterator()` ritorna uno `Spliterator<T>` che può essere attraversato **sequenzialmente** col metodo `tryAdvance(Consumer<? super T> action)`. Quest'ultimo ritorna `true` se c'è un elemento sul quale applicare la funzione. Quindi mettendo tutto dentro a un `while`, sto facendo "finchè ci sono elementi da analizzare (`tryAdvance()` ritorna `true`) applica l'operazione che ho specificato (usa `action`)". Altre azioni che uno *splititerator* permette sono:

- stimare gli elementi rimanenti con `long stimateSize()`
- impostare delle *characteristics* alla sorgente
- suddividere l'iterazione in più rami con `Splititerator<T> trySplit()`; che ritorna `null` se non è possibile suddividere in più rami l'esecuzione

Con l'utilizzo di `tryAdvance()` scorro la lista sequenzialmente ed analizzo gli elementi uno ad uno. Usando invece il `trySplit()` (che è il vero scopo per cui esiste lo `Splititerator`) si prova a dividere l'iteratore in due parti che verranno eseguite in parallelo. Esempio:

```
public static List<String> generateElements() {
      return Stream.generate(() -> new String("prova"))
            .limit(15)
            .collect(Collectors.toList());
}

//Poi nel main...
Spliterator<String> primo = generateElements().spliterator();
Spliterator<String> secondo = primo.trySplit();
```

Il metodo statico semplicemente ritorna una lista contenente la parola *prova*. Notare che nel secondo blocco c'è la chiamata `primo.trySplit()` che divide in due lo split iterator. Adesso è possibile iterare su primo e su secondo in modo parallelo:

```
@Test
assertThat(new Task(split1).call())
            .containsSequence(generateElements().size() / 2);
assertThat(new Task(split2).call())
            .containsSequence(generateElements().size() / 2);
```

Ignoriamo il `Task` che è solo ai fini di esempio (un `Task` è un thread che viene eseguito "sul posto"). Vediamo che partono in parallelo due esecuzioni indipendenti e alla fine verrà fuori che ciascuna contiene esattamente metà elementi dello *splititerator* di partenza. In altre parole, la `trySplit()` mi ha diviso in due lo `Splititerator<T>` iniziale e mi ha lasciato con due pezzi (due metà) che posso gestire indipendenti.

Gli stream quindi sono un'ottima astrazione per modellare semplicemente algoritmi su collezioni di elementi e possono essere attraversati in sequenza o in parallelo. In particolare, lo stream decide da solo "quanto parallelismo usare" attraverso il `ForkJoinPool` (un contenitore di Task, ovvero azioni in parallelo). Il **blocking factor** è l'intensità di calcolo di un algoritmo e può essere:

- `BF = 0`. L'algoritmo occupa costantemente la CPU
- `BF = 1`. L'algoritmo è costantemente in attesa di I/O

Il numero di thread è che possono essere impiegati per il parallelismo può essere ottenuto dalla formula:

$$NumeroThread = \frac{NumeroDiCores}{1 - BF}$$

Se l'algoritmo usa molta I/O si può modificare l'impostazione del pool per aumentare il numero di thread disponibili

PRIMITIVE DI COMUNICAZIONE DI RETE

La JVM fornisce supporto per le operazioni di rete attraverso il protocollo TCP/IP grazie ai *socket* ed i *datagram*. Si tratta del livello più basso di astrazione perché ci tocca lavorare direttamente con i bytes trasferiti e gestire tutto praticamente (connessione, errori, buffer...). A livello pratico e di lavoro solitamente queste non si usano ma si usano librerie HTTP che facilitano l'uso della comunicazione. Comunque tali librerie usano alla base queste primitive e quindi vale la pena vedere qualcosa.

Bisogna notare che per quanto riguarda il protocollo TCP/IP Java fa affidamento sui socket, mentre per il protocollo UPD si appoggia ai datagram. Ricordiamo che:

- TCP/IP: è il protocollo di rete più sicuro per la trasmissione dei dati che riporta anche conferma di ricezione. IP è il protocollo di routing che spezza i dati in pacchetti e li manda ad un indirizzo sulla rete (però non garantisce che arrivino tutti a destinazione, ci potrebbero essere nodi rotti nel mezzo) TCP è più ad alto livello e si occupa di mettere assieme i pacchetti e ritrasmetterli se necessario. Si usa per fare I/O su stream sicuro attraverso la rete.

- UDP: sta a fianco del TCP e può essere usato direttamente per una veloce comunicazione, senza connessione però con un inaffidabile trasporto dei pacchetti. I dati vengono inviati e chiunque sia in "ascolto" li prende ma chi li invia non si preoccupa degli errori. Invia e poi quel che succederà non è affare suo (comunicazione di tipo "broadcast").

Java utilizza la classe `InetAddress` per incapsulare gli indirizzi IP sulla rete e il nome di dominio per quell'indirizzo. Vediamo ora le due primitive di comunicazione che ci sono in Java, che servono ad implementare i protocolli di base per la comunicazione via internet

- **SOCKET TCP/IP**

  In Java la classe `Socket` è l'astrazione per la comunicazione bidirezionale (posso richiedere e ricevere dati da uno stesso canale), persistente e sicura fra due sistemi. Ci sono due tipi di TCP socket: il `ServerSocket` che è un "listener", ovvero aspetta che qualcuno si connetta prima di fare qualcosa, ed il `Socket` che è il client, ovvero colui che si connette al server per lo scambio di dati. Entrambe implementano l'interfaccia `AutoCloseable` quindi supportano il try-with-resources.

  La creazione di un socket solitamente avviene attraverso l'impiego del costruttore che accetta un indirizzo IP ed una porta; la connessione viene stabilita implicitamente.

  ```
  Socket(InetAddress ipAddress, int port) throws IOException
  ```

  Si può ottenere accesso agli stream di input ed output di un socket usando i suoi due metodi:

  ```
  InputStream getInputStream() throws IOException;
  OutputStream getOutputStream() throws IOException;
  ```

L'eccezione viene lanciata se lo stream è stato invalidato a causa di una perdita di connessione col server al quale sono collegati. Si usano come normali stream nei quali posso fare I/O, sono **thead safe** però soltanto uno per volta può leggere o scrivere. I buffer inoltre sono limitati però posso scegliere io la dimensione. Lettura e scrittura possono bloccare il thread. Il problema è che lo stream mi espone solamente i dati e basta, non mi dice se la richiesta è stata terminata e tanto meno mi segnala che tutta la risposta è stata inviata. Devo fare tutto io a mano.

Vediamo un esempio. Andremo a creare un `Socket` che si connetterà ad un server sulla rete reperibile all'indirizzo whois.internic.net e in ascolto sulla porta 43. Questo server riceve un url e mi ritorna un po' di informazioni relative a quel sito.

```
//Apro il socket --> creo la connessione
try (Socket s = new Socket("whois.internic.net", 43)) {

    //Ottengo gli stream per input ed output
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    //Creo la richiesta
    String req = "oracle.com\n";
    byte buf[] = req.getBytes();

    //invio la richiesta
    out.write(buf);

     //Leggo la risposta
     int c;
     while ( (c = in.read()) != -1 )
         System.out.print( (char) c );

}
```

Usando il try-with-resources mi assicuro che la risorsa (il socket `s`) venga liberata dopo che è stato utilizzata. La prima cosa che faccio è prendere i due stream per fare input ed output; sono loro che fanno da "tramite" fra me e il server perché si occupano di mandare e ricevere dati.

1. Voglio informazioni sul sito della oracle e quindi creo la variabile `req = "oracle.com\n"` ma poi la converto in un array di byte, poiché gli stream lavorano solo a byte
2. Mando al server la richiesta usando l'`OutputStream` che potrebbe bloccare
3. La risposta mi viene data come una serie di bytes da leggere. Nel `while` leggo tutti i byte (mi ritorna -1 quando arrivo alla fine) disponibili e poi li converto in carattere

Se non avessi utilizzato il try-with-recources, avrei dovuto chiamare `close()` dopo del `while`. Notare che sto usando gli stream normalmente quindi tutti i discorsi che valgono per stream I/O su stream valgono anche qui. L'output (un pezzo solo perché è lungo) è:

```
Domain Name: ORACLE.COM
Registry Domain ID: 607513_DOMAIN_COM-VRSN
Registrar WHOIS Server: whois.markmonitor.com
Registrar URL: http://www.markmonitor.com
Updated Date: 2018-10-30T09:24:31Z
Creation Date: 1988-12-02T05:00:00Z
```

Il `ServerSocket` come costruttore prende una porta sulla quale attendere richieste ed un indirizzo IP ad esso associato.

```
ServerSocket(int port) throws IOException
ServerSocket(int port, InetAddress address) throws IOException
```

Verrà poi chiamato il metodo **bloccante** `accept()` sul server che aspetta la connessione da parte di un client. La segnatura è `Socket accept()` perché viene ritornato il client al quale il server si è "attaccato" e userà i suoi stream per la comunicazione.

- **DATAGRAM**

I datagram sono implementati sul protocollo UDP il quale invia ogni singolo pacchetto verso una o più destinazioni. Quando un pacchetto viene mandato non si è sicuri che arrivi a destinazione o ci sia qualcuno che è in attesa. Non si sa nemmeno se sia arrivato il pacchetto integro!

Tuttavia questa comunicazione non è comparabile a TCP/IP in termini di velocità (UDP è velocissimo nella trasmissione) anche perché non si preoccupa delle congestioni o dell'integrità dei pacchetti. Java usa due classi:

1. `DatagramPacket`: è il contenitore dei dati di dimensione massima 64Kb

2. `DatagramSocket`: è il meccanismo usato per inviare o ricevere i `DatagramPacket`

Il costruttore di `DatagramSocket` ha diversi overload ma il più usato solitamente è quello che prende in ingresso una porta ed un `InetAddress`. A partire dalla JDK 7 questa classe implementa `AutoCloseable` per il supporto al try-with-resources. I due metodi più importanti sono:

```
void send(DatagramPacket packet) throws IOException
void receive(DatagramPacket packet) throws IOException
```

Il primo metodo manda il pacchetto all'indirizzo ed alla porta specificati mentre l'altro metodo aspetta un pacchetto in arrivo che deve essere ricevuto. Qui però a differenza dei `Socket` abbiamo a disposizione la dimensione del pacchetto e possiamo mandare un singolo messaggio a più indirizzi (modalità multicast, non supportata da socket perché è un protocollo a singolo canale).

Di contro, rispetto ad un socket perdiamo di affidabilità (nessun segnale di consegna). Vediamo un esempio di client e server:

```
class Server implements Runnable {

    //Necessario per inviare e ricevere i pacchetti
    private DatagramSocket socket;

    public Server(int port) throws SocketException {
        socket = new DatagramSocket(port);
    }

    @Override
    public void run() {
        //Buffer nel quale salvare i dati dei pacchetti
        byte[] buf = new byte[256];
        DatagramPacket pk = new DatagramPacket(buf, buf.length);

        //Mi metto in attesa di ricevere dei dati
        try {
            socket.receive(pk);
```

```
            var d = new String(pk.getData(), 0, pk.getLength());
            System.out.println("Ricevuto: " + d);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            socket.close();
        }
    }

}

public class Main {

    public static void main(String[] args) throws Exception {

        //Faccio partire il server su un nuovo thread
        new Thread(new Server(58325)).start();

        //Ai fini dell'esempio, aspetto un po' dopo l'avvio del
        //server e poi faccio partire un pacchetto dal client
        Thread.sleep(200);

        //Preparo un messaggio da mandare al server
        DatagramSocket socket = new DatagramSocket();
        byte[] buf = "Roberto".getBytes();

        //Mi preparo all'invio dicendo al pacchetto dove andare
        InetAddress address = InetAddress.getByName("localhost");
        DatagramPacket packet = new DatagramPacket(buf,
buf.length, address, 58325);

        //Invio il dato e mi ricordo di chiudere la connessione
        socket.send(packet);
        socket.close();

    }
}
```

Nel server creo un `DatagramSocket socket` che mi serve a gestire la connessione e poi nel corpo del thread, facendo `socket.receive(pk)`, sto dicendo "voglio ricevere un pacchetto `pk` di dimensione tot (specificata da `buf`)".

Siccome sto ricevendo byte, costruisco una nuova stringa che convertirà i bytes ricevuti in stringa e me li mostrerà a schermo. Il client crea anche lui un `DatagramSocket socket` perché senza di questo non posso comunicare. Converte la stringa *Roberto* in bytes (così può essere spedita) e poi chiama la `send(packet)`.

Java fornisce queste primitive di comunicazione che sono molto a basso livello e richiedono molto lavoro. Ci sono tuttavia anche altre classi utili come ad esempio `URL` il quale rappresenta un puntatore ad una "risorsa" nel web. Le due più comuni richieste che posso fare sono la GET e la POST.

- **GET**

  Al giorno d'oggi è molto utile che la nostra app o programma possa ricevere dati da un server; solitamente si fa una richiesta GET per ottenere del JSON con i dati di risposta. Per esempio:

1. Voglio sapere il calendario delle corse di Formula 1 nel 2018.
2. Nel programma faccio una richiesta GET a https://ergast.com/api/f1/2018.json e questa pagina mi ritorna un json coi dati delle gare del 2018 (ergast è un sito con le API per la Formula 1)
3. Nel programma faccio il parse del JSON e ottengo i dati

Il codice per ottenere questo JSON di esempio lo riporto qua sotto. La parte interessante è la creazione dello stream tramite `url.openStream()` che ritorna un `InputStream` contenente i dati ricevuti dalla pagina.

```
//Imposto l'url dal quale prendere i dati
URL url = new URL("https://ergast.com/api/f1/2018.json");

//Leggo il contenuto ritornato dalla pagina
BufferedReader reader = new BufferedReader(
        new InputStreamReader(url.openStream())
);

//Stampo a schermo
String line;
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}
```

- **POST**

Questo verbo http invece di ricevere dati li invia facendo quindi l'opposto di quello che fa il GET. Si può utilizzare ancora la classe URL. Vediamo un esempio:

```
//Creo l'URL con l'indirizzo nel quale voglio fare la POST
URL url = new URL("https://httpbin.org/post");
URLConnection connection = url.openConnection();

//Devo mettere questo flag a true se voglio fare la POST!
connection.setDoOutput(true);

//PrintWriter è sottoclasse di Writer e serve a stampare testo
//formattato su uno stream di output
PrintWriter writer = new PrintWriter(
    connection.getOutputStream()
);

//Faccio la post
writer.println("test=val");
writer.close();
```

In questo modo ho mandato una POST all'indirizzo https://httpbin.org/post e, se volessi leggere la risposta che mi manda il server, potrei usare il codice sopra chiamando

```
BufferedReader reader = new BufferedReader(
        new InputStreamReader(url.openStream())
);
```

e mostrando su terminale la risposta.

Per completare la panoramica, avendo visto come scrivere un server ed un client usando rispettivamente le classi `ServerSocket` e `Socket`, vediamo anche come creare un server che gestisca richieste asincrone con l'utilizzo dei channels. Andremo ora ad analizzare due classi che sono state introdotte in Java 1.4 all'interno del package `java.nio`. In particolare ci sono due interfacce da considerare:

```
interface Channel extends Closeable { /* ... */ }
```

Un Channel rappresenta un canale di I/O che può essere aperto (per consentire lo scambio di dati) o chiuso (non permette di fare I/O). Esiste anche una versione di `Channel` in grado di comunicare sulla rete, ovvero

```
interface NetworkChannel extends Closeable { /* ... */ }
```

e rappresenta un canale di comunicazione attraverso la rete. Notare che entrambe implementano `Closeable` e quindi posso usare il try-with-resources. Vediamo ora le due classi principali che ci interessano per quanto riguarda la comunicazione asincrona.

Prima di parlare di server asincroni, conviene mostrarne uno sincrono per caprie cosa cambia. Quando scrivo un server **non** asincrono, questo vivrà all'interno del `main` o comunque dentro allo scope di qualcuno e sarà bloccante. Vediamo un esempio veloce:

```
public static void main(String[] args) throws IOException {
      //0. Mi permette di selezionare i client che si connettono
      Selector selector = Selector.open();

      //1. Creo il server su una certa porta ad un certo indirizzo
      ServerSocketChannel serverSocket = ServerSocketChannel.open();
      InetSocketAddress sA = new InetSocketAddress("localhost", 1111);

      //2. "Collego" il server a quell'indirizzo
      serverSocket.bind(sA);
      serverSocket.register(selector, serverSocker.validOps(), null);

      //3. NECESSARIO per mantenere il server attivo
      while (true) {
            //uso il server....
            selector.select();
      }
}
```

Qui il `ServerSocketChannel` ha bisogno di `selector` (vedi capitoli sopra, la parte su NIO) che in pratica **blocca** il thread fino a che almeno un client si connette. Quindi `selector.select();` è una chiamata bloccante che aspetta una connessione; quando questa arriva allora procede.

Cosa succede se la `selector.select();` trova più di una connessione (cioè più di un canale è pronto a fare I/O)? Semplicemente si fa un ciclo che, uno alla volta (cioè **sequenzialmente**), processa tutte le richieste. Vediamo ora come poter fare queste cose ma in parallelo.

AsynchronousServerSocketChannel

```
public abstract class AsynchronousServerSocketChannel
  implements AsynchronousChannel, NetworkChannel
```

Si tratta di un canale asincrono bastato su un server socket; detto in altre parole, è un server che si basa sui `Socket` ma che non blocca il thread principale e può gestire in modo asincrono (in parallelo) le connessioni. Nell'esempio di prima avevo la chiamata a `select()` che bloccava e dovevo aspettare che uno o più client fossero pronti (ed eseguire in sequenza le richieste).

Adesso usando `AsynchronousServerSocketChannel` posso gestire tutte le richieste in parallelo. Se quindi mi arrivano tante richieste assieme, al posto di eseguirle una ad una in un ciclo le esegui tutte **assieme** (in parallelo) ognuna nel suo thread. È un grande vantaggio, anche perché non ho bisogno di cose tipo `selector.select();` perché questo tipo di server non blocca; ritorna subito e lascia "libero" il main (o comunque il thread che lo sta usando) per fare altre cose.

Come abbiamo visto quando parlavamo di thread, quando un thread parte in parallelo non abbiamo più controllo su di lui; non sappiamo quando la JVM deciderà di farlo partire e quando terminerà. Per questo, quando un `AsynchronousServerSocketChannel` esegue una richiesta in parallelo ha bisogno di questo:

```
interface CompletionHandler<V, A>
```

Questa interfaccia va implementata e indica l'azione da eseguire dopo che una richiesta parallela è stata eseguita. Siccome **NON** so quando la richiesta verrà del tutto completata (perché in parallelo) mi serve questo `CompletionHandler` che mi notifica "guarda che ho finito sta cosa in parallelo, adesso dimmi cosa fare". Vediamo un esempio di codice molto semplice:

```
//Creo un AsynchronousServerSocketChannel e gli assegno un indirizzo
//e una porta sulla quale ascoltare (con InetSocketAddress)
final AsynchronousServerSocketChannel listener =
      AsynchronousServerSocketChannel.open().bind(
          new InetSocketAddress(5000)
      );

//Adesso specifico cosa fare quando arriva una nuova connessione
  listener.accept(null, new
CompletionHandler<AsynchronousSocketChannel, Void>() {
      public void completed(AsynchronousSocketChannel ch, Void att) {
          //...
      }
      public void failed(Throwable exc, Void att) {
          //...
      }
  });
```

Nella prima parte, molto semplicemente, creo il server. Poi nella seconda parte accetto la connessione d dico cosa fare una volta che ho terminato l'operazione o nel caso ci siano stati errori. Nel particolare:

- `public abstract <A> void accept(A attachment, CompletionHandler<AsynchronousSocketChannel, ? super A>)`

  Dico al server cosa fare quando si connette ad esso un client. Questo metodo fa partire un'operazione asincrona (= su un altro thread) e prende due parametri in input:

  - Il primo prende l'oggetto sul quale voglio fare le operazioni di I/O e può anche essere `null` se non devo passare niente. Questa cosa mi permette di far "circolare" dentro fra i thread (in modo sicuro) l'oggetto di I/O.
  - Il secondo parametro è il famoso `CompletionHandler` che verrà invocato quando il thread parallelo ha processato il risultato

- `void completed(V result, A attachment)`

  Questo metodo viene chiamato quando l'operazione di I/O richiesta è stata eseguita dal server con successo, ovvero senza eccezioni

- `void failed(Throwable exc, A attachment)`

Questo metodo viene chiamato quando l'operazione richiesta **NON** è stata eseguita dal server con successo perché ci sono stati errori. Tramite l'uso di `exc` posso fare debug.

Riassumendo: con `AsynchronousServerSocketChannel` creo un server asincrono, poi col metodo `accept` definisco cosa bisogna fare una volta che la connessione è stata stabilita. Per definire cosa fare dopo l'operazione di I/O uso il `CompletionHandler` (coi suoi due metodi da definire). Il parametro **attachment** permette di fare "circolare" le informazioni di contesto nei thread.

In alternativa all'uso di un `CompletionHandler` si può anche utilizzare una versione di `accept()` che ritorna un dato di tipo `Future`.

```
public abstract Future<AsynchronousSocketChannel> accept()
```

In pratica quando faccio la `accept()` ritorna subito e con l'oggetto future che ho potrò chiamare il `get()` che però potrebbe essere bloccante. Con questo approccio si perde anche l'essenza del server asincrono perché non ho più garanzia di non bloccaggio.

```
Socket socket = new Socket("localhost", SERVER_PORT);
```

Per connettersi ad un `AsynchronousSocketChannel` posso usare un normalissimo `Socket` e connetterlo all'indirizzo del server. Non ho necessità di una classe asincrona per il socket perché tanto tutta la parte di parallelismo deve farsela il server. È il server che può ricevere tante richieste in contemporanea ma il socket manda sempre una ed una sola richiesta. Se mi dovesse servire un socket parallelo, ne faccio io uno di custom usando `Thread` o la primitiva che mi serve.

Il vero modo di costruire un server per la produzione tuttavia è quello di usare dei **web framework** perché gestire un sistema complesso a mano con primitive a basso livello (i socket, datagram etc) è difficile e ci vuole un sacco di tempo. Le astrazioni dei framework semplificano la vita e permettono spesso di adattarsi a più soluzioni.

Un esempio è quello **`Vert.X`** ovvero un framework creato dalla Eclipse Foundation (quelli che fanno l'IDE per Java) ed è scalabile, event-driver, asincrono e funziona con più linguaggi. Si occupa della creazione del server, degli indirizzamenti, della gestione delle pagine; fa tutto lui.

```
//Creo un'istanza di Vert.X per il setup del server
Vertx vertx = Vertx.vertx();
var options = (new HttpServerOptions()).setLogActivity(true);

//Creo l'istanza del server HTTP
HttpServer server = vertx.createHttpServer(options);

//Creo il gestore degli indirizzi e delle richieste
Router router = Router.router(vertx);

  //Quando apro www.sito.com/ il server risponde con la homepage
  router.get("/").produces("text/html").handler(ctx -> {
    //uso ctx per mostrare la pagina html di index
  });

  //Quando apro www.sito.com/ il server risponde con la homepage
  router.get("/welcome").produces("text/html").handler(ctx -> {
    //uso ctx per mostrare la pagina welcome.html o quello che voglio
  });
```

In 4 righe ho creato il server e non devo fare altro perché si occuperà di tutta la gestione degli eventi, in modo asincrono, il framework. Poi Ho creato un `Router` che è l'astrazione di Vert.X che consente di associare ad un URL del sito una lambda che dirà cosa fare.

In questo caso avendo messo `router.get("/").produces("text/html")` sto dicendo che quando andrò ad aprire la home page del sito, verrà prodotta una pagina HTML. Nella lambda usando la variabile `ctx` posso caricare un file html chiamato index.html su disco, posso scrivere una stringa o fare quello che voglio.

- Questo framework ha il tipico approccio dei sistemi asincroni: viene fornito un handler che verrà eseguito in modo asincrono al verificarsi di una certo evento (event-driven async framework)
- La gestione coerente della condivisione dei dati è a nostro carico
- Rende facile specificare gli indirizzi (grazie a `Router`) e costruire le risposte da dare e rende trasparente la gestione della sicurezza e dei dettagli del protocollo

Tuttavia un framework può rendere difficile la gestione di specifiche richieste di client che vanno fuori da quelli che sono gli standard di comunicazione e non mi consente di controllare al 100% l'erogazione della risposta al client. Questo perché comunque è un framework e la maggior parte del lavoro lo fa il codice; io posso solo avere potere su quello che il framework mi da per interfacciarmi (tipo i metodi setter o la specifica degli handler) ma non posso fare di più.

ESTENSIONI DEL LINGUAGGIO JAVA

Abbiamo visto tutti i più importanti metodi e paradigmi per affrontare la programmazione concorrente in Java, sono davvero tanti. Però ci sono anche tanti altri progetti esterni (non presenti nei package di java) che supportano la programmazione concorrente.

- **Project Loom**

  Insieme di features per Java che aggiunge il concetto di *Fiber* che sarebbe una versione più leggera di un thread java. Si cerca di poter gestire migliaia di *fiber* assieme per elaborare compiti onerosi parallelizzabili e che magari possono anche fare I/O. Si pone quindi come alternativa a `Thread` e tutto quello che ci va dietro.

- **Reactive extensions**

  Gli stream di Java si basano sul concetto di *Inversion of Control* che è già stato visto perché l'onere dell'esecuzione (come fare le varie operazioni) spetta la framework, non al programmatore. Le *Reactive Extension* forniscono un modello di esecuzione per elaborare sequenze di oggetti in modo asincrono. Si fondano sull'**observer pattern**.

  Rispetto algi stream fonriscono una semantica più ricca, più regolarità di composizione ed una indipendenza dal modello di esecuzione (sincrono o asincrono). Viene introdotta la gestione della **backpressure**, ovvero l'impedimento che possano accadere dei problemi a causa di un sovraccarico di dati ricevuti da un'unità lavorativa. Serve quindi a mantenere equilibrio

- **AKKA**

  Porta il modello ad attori di Java nel 2011; ci sono più entità che comunicano fra di loro scambiandosi messaggi e basta. Un attore è un'entità composta da: stato (che è privato dell'attore), comportamento (cosa fare in risposta ad un messaggio), mailbox (un attore riceve i messaggi in una mailbox), figli (un attore può creare altri figli e farli lavorare concorrentemente).

JAVA E LA DISTRIBUZIONE

La tecnica di programmazione distribuita si riferisce ad un insieme di modi per poter gestire più processi su macchine diverse che operano in modo coordinato; una sorta di parallelismo "multi computer". Fino ad

adesso abbiamo visto come poter i vari core di un singolo computer ed impegnarli facendo partire i thread in parallelo e tutte le storie varie.

Un insieme di elaboratori che eseguono un algoritmo in modo distribuito è detto **sistema distribuito**. Le principali motivazioni che hanno portato ad ideare la distribuzione di un algoritmo su più nodi sono:

1. **Affidabilità.** Le attività possono proseguire sui nodi che sono ancora "vivi" rispetto a quelli che hanno avuto errori oppure sono fermi per problemi.

2. **Suddivisione del carico**. Una mole di lavoro più grande delle capacità di una sola macchina può essere suddivisa fra più nodi per essere eseguita in modo concorrente.

3. **Distribuzione**. Gli utenti di più macchine possono accedere ai risultati del lavoro da uno qualsiasi dei nodi.

Non tutti i tipi di algoritmi possono girare su sistemi distribuiti perché ci sono diversi fattori da tenere in considerazione, inclusi fra questi anche i tre di sopra. In particolare, un *algoritmo distribuito* deve avere queste caratteristiche:

1. **Concorrenza dei componenti.** I vari nodi di esecuzione (= i vari pc sparsi in giro ma collegati in rete) operano in modo concorrente quindi l'algoritmo in questione non deve soffrire di problemi legati all'aspetto del multithreading.

2. **Mancanza di *global clock***. L'ordine temporale degli eventi non è condiviso dalle macchine in quanto ognuna lavora secondo certe regole. I pc possono non essere uguali e quindi il modo di gestire l'algoritmo (in particolar modo se con i thread) può variare. Se l'ordine temporale degli eventi è importante, va impostato "a mano".
3. **Fallimenti indipendenti**. I nodi possono guastarsi o fallire indipendentemente uno dall'altro.

I nodi comunicano fra loro scambiandosi messaggi perché hanno a disposizione solo la rete per parlare fra loro. L'astrazione principale di comunicazione quindi di basa sull'invio di un messaggio e l'attesa della ricezione di un altro messaggio o di un alert.

Il termine **RPC** (*Remote Procedure Call*) indica un sistema per far sembrare che le chiamate ai metodi avvengano sul computer locale invece che sulla rete ad un altro computer. L'adattamento locale/remoto viene implementato da un componente detto **stub**. Quando devo chiamare una funzione accade ciò (la risposta segue il processo inverso di quello descritto):

- Il client fa una chiamata allo *stub* locale
- Lo *stub* mette i parametri della funzione (se ce ne sono, sennò niente) all'interno di un messaggio
- Lo *stub* invia il messaggio al nodo di destinazione attraverso la rete
- Sul nodo di arrivo c'è un server *stub* che sta in ascolto dei messaggi e quando ne arriva uno, allora sa che c'è stata una chiamata alla funzione.
- Il server *stub* estrae i parametri e fa la chiamata alla procedura locale

Quando si parla di OOP tutto questo meccanismo prende il nome di **RMI** (*Remote Method Invocation*) in quanto il messaggio deve includere anche l'indirizzamento dell'oggetto destinatario della chiamata. Avviene quindi l'invocazione di un metodo di un oggetto in remoto (= invoco un metodo di un oggetto che sta in un pc sulla rete) sembrando che questo oggetto sia "locale", cioè sullo heap del mio pc.

Negli anni '90 diventa popolare CORBA, una tecnologia di RMI che descrive gli oggetti tramite un linguaggio particolare chiamato IDL. Java implementa fin da subito un sistema di RMI ed abbraccia CORBA. In Java una chiamata RMI comporta degli step simili a quelli visti per RPC però con delle aggiunte:

- Il client fa una chiamata allo *stub* locale
- Lo *stub* mette i parametri della funzione (se ce ne sono, sennò niente) all'interno di un messaggio
- Lo *stub* invia il messaggio al nodo di destinazione attraverso la rete. **Il client è bloccato**

- Sul nodo di arrivo c'è il server (chiamato *Object Broker*) che riceve il messaggio, lo controlla e cerca l'oggetto chiamato
- Il server recupera i parametri e chiama il metodo che si vuole.
- L'oggetto esegue il metodo, ritorna.

Il server ripete tutto da capo chiamando lo *stub* locale per inviare la risposta ed il client, una volta ricevuta la risposta, **si sblocca**. Il problema è che in questo lungo processo un sacco di cose possono andare male: messaggi non recapitati, client/server hanno oggetti con segnature diversi a causa di aggiornamenti fatti col culo, problemi di rete o il server che muore.

> Al giorno d'oggi le tecnologie RMI sono poco diffuse; in Java 9 il modulo `java.corba` è stato deprecato per poi essere completamente rimosso in Java 11

Un aspetto fondamentale del sistema RMI è quello della serializzazione, ovvero il modo in cui viene predisposto un oggetto per la trasmissione di un messaggio. Come è già stato visto, la **serializzazione** consente in poche parole di convertire un oggetto di un linguaggio in una qualche forma (tipo un json o un XML) che lo rappresenta. Poi da qui è possibile fare la **deserializzazione** che a partire da questa forma riesce a costruire l'oggetto originale.

Per far comunicare fra loro nodi distribuiti conviene usare le classiche primitive del modello TCP/IP (`Socket` per TCP e `Datagram` per UDP), l'astrazione di Channel per unificare le operazioni di I/O su canali differenti (file, rete, hardware) e le varie utilità offerte da `java.nio` per sfruttare le potenzialità del SO. Da ricordare anche la classe `URL` per semplici richieste sul protocollo http.

Vediamo ora gli 8 principali problemi che si possono incontrare quando si decide di utilizzare una struttura distribuita per la creazione di un sistema:

1. **Network is not reliable**

   Una rete può essere inaffidabile per un sacco di motivazioni diverse, come la rottura di un computer, un software con bug oppure semplicemente a causa della rottura di un filo

2. **Latency is zero**

   C'è un limite fisico alla velocità di trasmissione di un messaggio (velocità della luce) e quindi in sistemi in cui il tempo è una criticità bisogna considerare questo. Ad esempio fra le due coste dell'atlantico è impossibile raggiungere una latenza inferiore ai 30ms!

3. **Bandwidth is infinite**

   La banda disponibile è sempre meno un problema ma i dati da trasferire aumentano più del tasso di miglioramento della qualità della banda. In molte situazioni alcuni protocolli di rete danno priorità all'affidabilità della comunicazione invece che al costo della trasmissione

4. **Newtork is secure**

   Gli h4ck3r cattivi che cercano dei bug all'interno dei messaggi o del sistema di trasporto che possa fornire delle vulnerabilità. Per questo servono autenticazione, autorizzazione e credenziali di accesso alla rete

5. **Topology doesn't change**

   Siccome i nodi sparsi nella rete possono spostarsi e cambiare indirizzo, non possiamo assumere che la mappa dei nodi sia sempre quella. La topologia cambia e quindi bisogna che il sistema sia in grado di adattarsi ai cambiamenti e quindi che possa aggiornare gli indirizzi per raggiungere i nodi

6. **There is one administrator**

   La diffusione e frammentazione delle reti significa anche frammentazione delle responsabilità nella gestione delle reti stesse. Diverse organizzazioni possono essere coinvolte nella gestione del sistema e bisogna tenere conto di questo nell'implementazione dei dettagli di sistema

7. **Transport cost is zero**

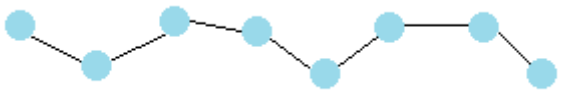
   Anche il costo del trasporto sta continuamente calando, ma non è mai zero. Sia in termini monetari, sia anche in termini energetici: la maggior parte dei dispositivi è a batteria

8. **The network is homogeneous**

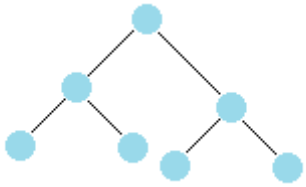
   Le reti sono formate da un misto di tecnologie ed i pacchetti attraversano una vasta serie di mezzi trasmissivi, metodi di comunicazione ed apparati differenti. Per questo motivo il protocolo di comunicazione non deve avere particolari esigenze di trasporto; il dato quindi deve poter viaggiare dappertutto ed arrivare a destinazione qualunque sia il mezzo che lo porta

Abbiamo quindi giustificato la necessità di realizzare un sistema distribuito con la ricerca di affidabilità, suddivisione del carico di lavoro e distribuzione dei risultati. Anche i dati stessi necessitano delle stesse caratteristiche. Un insieme di dati gestiti da un **sistema distribuito** è disponibile, accessibile e coerente solo se tutti i nodi riescono a stare "allineati", cioè sincronizzati e collegati fra di loro.

Il problema è che se ho miliardi di nodi non posso avere un sistema lineare sennò ci vuole una vita per andare dal primo fino all'ultimo nodo; serve una struttura gerarchica o comunque qualcosa di meglio di una struttra a lista concatenata.



Per questi motivi, nel sistema a DNS (Domain Name System) ogni nodo è responsabile di un ramo di un albero di indirizzi. In altre parole, ogni nodo a sotto di lui una serie di altri nodi e c'è una struttura ad albero



nella quale ogni nodo ha sotto di lui altri nodi figli ma in questo modo, ad ogni passo di profondità che scendo escludo un sacco di altri nodi (quelli che prendono altri cammini). Un client chiede la risoluzione di un ramo di un albero degli indirizzi e la responsabilità viene suddivisa fra i nodi: il nodo che non ha risposta propaga la richiesta ai figli o a chi gli sta più vicino. In questo modo non serve analizzare tutto.

Tuttavia se il requisito principale è la resistenza ai fallimenti o le prestazioni particolarmente elevate (in spazio o velocità) non abbiamo altra scelta che replicare lo stato in più di un nodo, in modo da avere:

- Nessun problema nei guasti dei nodi → se un nodo si rompe ho una copia dei dati in altri nodi
- Nessun problema di spazio → basta aggiungere nodi per aumentare la capacità del sistema

Aumentando i nodi però aumenta anche la dimensione del problema del **consenso**, ovvero il fatto che tutti i nodi del sistema abbiano la stessa versione di un dato (cioè che tutti i nodi siano aggiornati). Questo problema è noto come il problema dei *generali bizantini.*

- Diversi generali di un esercito Bizantino assediano una città. Devono raggiungere un consenso unanime su di una decisione: attaccare o ritirarsi. Conviene attaccare tutti o ritirarsi tutti perché un attacco con metà truppe verrebbe male; il problema è: come invio i messaggi? Sono sicuro che arrivino? Sono sicuro che mi arrivi risposta?

Il problema modella il caso in cui un nodo di sistema distribuiti si comporti in modo diverso ad ogni risposta e non posso sapere dall'esterno se il suo comportamento sia giusto (o sbagliato in caso sia rotto). Che fare?

Esistono gli *algoritmi di consenso* come **PAXOS** che si basa su un protocollo a tre fasi in grado di risolvere questo tipo problema. Oppure c'è anche **RAFT** che si pone gli stessi obiettivi di PAXOS ma cerca di essere più facile da usare e compensibile. Tramite il consenso quindi possiamo ottenere che tutti i nodi siano aggiornati e quindi che il sistema distribuito abbia uno stato "coerente". Quali sono i limiti da prevedere in caso di guasti?

Vediamo un esempio concreto di sistema distribuito applicato ai database. Il **CAP** theorem definisce le tre caratteristiche di un database distribuito:

- **C – Consistency:** Ogni lettura di dati riceve il valore più recente o un errore
- **A – Availability**: Ogni richiesta rivece una risposta valida (ma non è detto che riceva il valore più recente, diversamente da C)
- **P** – **Partition tolerance:** Il sistema funziona anche se dei nodi muoiono perché rete viene partizionata

Il teorema afferma che solo due delle proprietà sopra possono valere contemporaneamente ma siccome ogni rete può avere guasti, la P vale sempre. Siccome la P vale sempre ed al massimo due condizioni possono essere verificate assieme, il teorema afferma che:

> In caso di partizione di rete un sistema può essere o consistente o disponibile

Valgono quindi AP o CP assieme ma **mai** CA. In altre parole un database distribuito, in caso di partizione perché ci sono stati guasti ai nodi o mi ritorna un valore però questo non è sempre l'ultimo che è stato scritto nel database oppure mi ritorna sempre il valore più recente ma in certi casi ho errore.

Spesso capita che un nodo abbia nuove informazioni da dare a qualcuno e che tali informazioni vadano riunite con quelle prodotte indipendentemente da un altro nodo. Non serve quindi il consenso su di un dato. Ogni nodo produce nuove versioni di uno stesso dato e ogni altro nodo del sistema deve aggiorarsi in base ai cambiamenti fatti dagli altri! In pratica se ho A, B, C collegati ed A produce una nuova versione del dato, allora B e C si devono collegare ad A e aggiornarsi in base a quello che A ha prodotto. Le due soluzioni a questo problema sono:

- **Operational Transformation.**

  Un nodo fa una modifica interna, le modifiche vengono propagate in broadcast agli altri nodi e ciascun nodo che riceve questo nuovo aggiornamento si adatta (cioè modifica i suoi dati interni in base a quelli ricevuti) → poco successo perché questa soluzione è difficile da implementare

- **CRDT.**

  Si tratta di un insieme di strutture dati. Una CRDT messa su di un sistema può essere modificata indipendentemente ma mi dà la garanzia che esiste un modo unico di aggiornare tutte le modifiche dei nodi risolvendo ogni conflitto di dati. In pratica quando aggiorno un nodo, questa roba qua mi assicura che anche tutto il resto che dipende da quel nodo viene sistemato

FINE