Paradigmi semplici (per davvero)

Rovesti Gabriel

Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.
Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

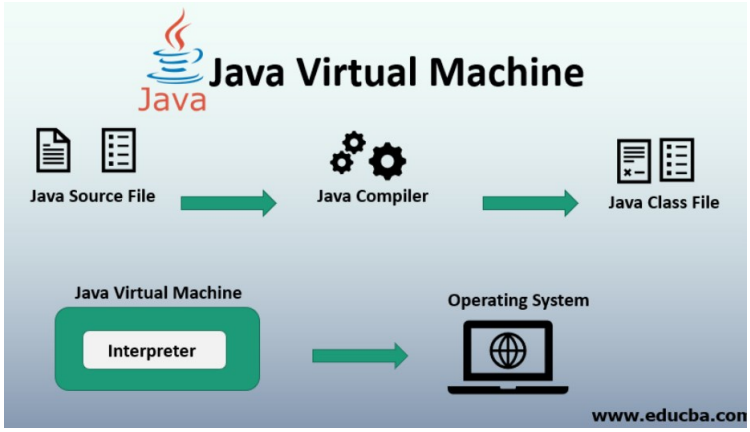
Lezione 1: Introduzione ai paradigmi, Java, JVM e concetti base

Si parla di *paradigma di programmazione* o paradigmi più in generale come una tecnica o un insieme di tecniche per affrontare una classe di problemi (ad es. Von Neumann, con la CPU).
In particolare, vedremo il paradigma della *concorrenza*, con più linee di esecuzione, asincrone, che condividono l'uso delle risorse in modo coordinato; contemporaneamente si parla di *parallelismo* con più linee di esecuzione che eseguono equivalentemente lo stesso calcolo su una partizione dei dati di ingresso (può essere considerato un sottoinsieme del caso concorrente; usa una parte degli stessi metodi).

Altro paradigma è l'esecuzione *in rete*, con un calcolo distribuito su nodi che condividono risorse e mezzi di comunicazione per mezzo di un'interfaccia di rete (Le altre linee di esecuzione non condividono più le stesse risorse (indipendentemente dal fatto che siano o meno sullo stesso nodo di esecuzione) e l'unico strumento è la comunicazione attraverso la rete. L'asincronicità è una necessità. In un certo senso è un'estensione della concorrenza).

Analogamente, la *distribuzione* prevede nodi indipendenti e, attraverso una rete non affidabile, coordinano l'esecuzione dello stesso lavoro in maniera consensuale su un certo sistema (Di questo paradigma vedremo solo alcuni risultati teorici: affrontarlo dal punto di vista pratico richiederebbe un intero corso partendo da una base di nozioni maggiore di quella richiesta qui. Ancora, è un'estensione della concorrenza e della comunicazione in rete).

In ultimo, si considera la *reattività*, costruito sulle basi della comunicazione asincrona tramite messaggi da cui ottiene caratteristiche di flessibilità, resilienza e scalabilità, rimanendo quindi affidabile e adattandosi nel corso del tempo (con "reattività" si intende una bassa latenza alla risposta; vale a dire, il sistema risponde sempre molto velocemente, anche in condizioni di utilizzo elevato delle risorse disponibili. Nasce dalla necessità di identificare, e rendere facili da realizzare, le condizioni in cui un sistema distribuito lavora in modo corretto ed efficiente)



Per cercare di studiare tutto ciò si usa Java, compilato attraverso il *bytecode*, linguaggio macchina intermedio interpretato dalla *Java Virtual Machine/JVM*. Esso è interpretato e volendo già compilato durante l'esecuzione (aka *JIT, Just In Time*) oppure prima della stessa esecuzione. La filosofia generale è *WORA* (Write Once Run Anywhere). Java si presta a tutti questi principi in linea di massima e di design, dato che la JVM stessa è sintomo di una idea di compilazione universale e semplice alle sue radici. Il linking della compilazione è dinamico, dato che guarda i nomi delle classi e dei metodi a tempo di compilazione/runtime.

Il caricamento del codice viene gestito da una gerarchia di ClassLoader, dividendo il caricamento delle classi di libreria da quelle di estensione, separando la visibilità di classi particolari e permettendo il caricamento del codice da sorgenti differenti dal filesystem, per esempio da un URL. In particolare, alla JVM è utile il *CLASSPATH*, quindi l'elenco delle locazioni in cui cercare una specifica classe.

Il comando *javac* invoca il compilatore e trasforma un file sorgente (di tipo *.java*) in un file bytecode (di tipo *.class*). I file sorgente devono chiamarsi come la classe in essi contenuta, il percorso delle directory deve corrispondere al loro package e, nei sorgenti o nel CLASSPATH, devono esserci tutti i tipi nominati dai sorgenti. L'ordine di compilazione, essendo una logica runtime, non è poi molto importante, dato che organizza tutto il compilatore stesso.

Il comando *java* avvia la JVM eseguendo il bytecode contenuto nel CLASSPATH; esso viene di norma contenuto dal file *.class*; similmente viene gestita anche l'archiviazione dei file e di codice Java con il comando *jar* che è stato per lungo tempo un formato di archiviazione utile, assieme a similari come *war* oppure *ear.* Utile notare che possono essere firmati per garantirne l'autenticità ed integrità.

Seguono alcune note storiche e commerciali: ricordiamo che Java è posseduta da Oracle e il software di tipo Java, per quanto in decadenza ormai, viene maggiormente utilizzato da fondazioni come Apache oppure Eclipse. Strumenti utili per costruzione e/o build usati anche in ambiti commerciali sono Apache Maven, che deve avere una chiara idea di come un progetto sia organizzato ed ogni libreria o componente può essere aggiunto ad un progetto indicandolo secondo delle coordinate: *gruppo:artefatto:versione*, oppure Gradle, che per la configurazioni non usa più XML ma un linguaggio apposito di programmazione noto come Groovy, con un approccio generalmente più dinamico.

Classi e tipi (1) - variabili, tipi, classi annidate, inizializzatori, tipi vari

In Java l'unità principale di esecuzione sono le *classi*, che possono essere dei metodi con nomi o eventualmente blocchi anonimi. Una classe appartiene ad un *package*, che organizza le classi in maniera gerarchica, il quale normalmente è la prima linea non commento di un pezzo di codice e corrisponde ad una linea DNS scritta in ordine inverso (se il package è assente, la classe di dice appartenente al package default, con nome pari alla stringa vuota. In caso di codice sperimentale o di prova non è un problema, ma alcuni strumenti o librerie vedono di cattivo occhio classi nel package di default; è assolutamente sconsigliato in caso di codice condiviso con altri o distribuito); alcuni package come *java*/*javax* sono riservati.

Se una classe appartiene allo stesso package, starà nella stessa directory oppure sta in un altro package e quindi sta da un'altra parte. La JVM può essere configurata per impedire l'accesso a determinati insiemi di classi, magari anche per motivi di sicurezza.
Dichiarando una classe senza modificatori di visibilità, la classe è visibile solo nello stesso package su cui si sta lavorando e non a classi esterne allo stesso.

Si passa alle *variabili*, ciascuno con un nome ed un suo tipo; esse definiscono la struttura di un oggetto di una classe. Vi sono due categorie di variabili:

1) di istanza, quindi ogni oggetto ha la propria copia e fa parte del suo stato
2) statiche, di cui ce ne sta una copia sola. Le variabili statiche vengono allocate ed inizializzate dal ClassLoader e preparate per l'uso.

Per esempio, la JVM non è obbligata a dare garanzie sull'ordine di inizializzazione delle classi, quindi JVM di versioni o produttori differenti potrebbero usare strategie diverse; oppure in alcune configurazioni molto particolari, la stessa classe potrebbe essere caricata da più Classloaders e quindi avere più copie delle variabili statiche.

Un esempio può essere:

```java
package it.unipd.pdp2021;

public class App {
  public static char c;
  int a;
  protected String internal;
  private boolean secret;
}
```

Modificatore	Classe	Package	Sottoclasse	Univ
public	✅	✅	✅	✅
protected	✅	❌	✅	❌
nessuno	✅	✅	❌	❌
private	✅	❌	❌	❌

Distinguiamo le variabili di tipo *public*, che sono visibili da qualsiasi altra classe caricata in cui, usando la direttiva *import*, si aggiunge al file sorgente il nome della classe importata.
La classe importata deve essere disponibile al momento della compilazione. Le direttive import devono trovarsi immediatamente dopo la direttiva package.
Se vengono usate due classi con lo stesso nome, una sola potrà essere importata; l'altra andrà richiamata per esteso. Il package java.lang si considera sempre importato.

Oltre alle variabili pubbliche, possono esserci le variabili *protected*, lette e scritte da classi che estendono la classe (cioè si usa *extends,* concetto di ereditarietà), le variabili *private*, visibile solo dal codice della classe stessa, *default*, se visibili solamente da classi dal package stesso.

Altri modificatori indicati sulle variabili sono *final*, variabile che non può più essere modificata dopo la costruzione (e quindi richiedono obbligatoriamente un valore in fase di esecuzione e potrebbe dare errore di esecuzione), *transient*, variabile ignorata in sede di serializzazione (oggi uso di questa ormai scomparso), *volatile*, regolando l'accesso concorrente alla variabile.
Nota: le variabili hanno per convenzione nomi in Camel Case (con lettera minuscola iniziale), tranne le variabili *static final*, il cui nome si scrive solitamente in maiuscolo, magari con parole separate da underscore. Le classi invece hanno la prima lettera maiuscola, quindi in Pascal Case.

La classe organizza il proprio codice in *metodi*, con alcuni possibili modificatori, un tipo di ritorno, un nome, un elenco di parametri ed opzionalmente un blocco di eccezioni oppure anche un blocco di codice da eseguire.

Con lo stesso nome della classe, solitamente ci sono i costruttori, chiamati quando si richiede la creazione di un oggetto specifico.

La tupla formata da nome metodo, parametri di tipo ed elenco dei tipi degli argomenti è detta *firma/signature* del metodo. Se non si ritorna nessun valore, si avrà un ritorno di tipo *void*. Il metodo dichiarato come statico è legato alla stessa classe; non può essere richiamato su un oggetto e non ha accesso alle variabili di istanza.

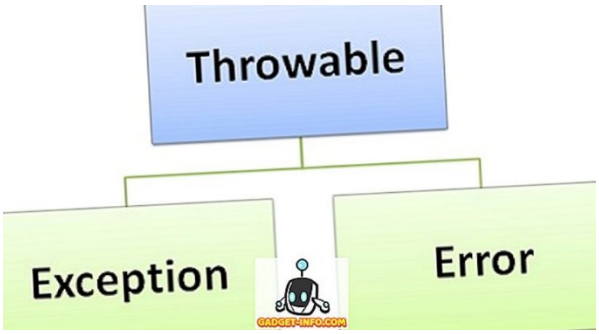
```java
package it.unipd.pdp2021;

public class App {
  public App() { };

  int apply(char d) { return 0; }

  static boolean prepare(String target, int count)
    throws RuntimeException {
    return false;
  }
}
```

Essi sono chiamati con la notazione *valore.nomemetodo(parametri)* (tecnicamente, una classe può avere una variabile ed un metodo con lo stesso nome: il compilatore è in grado di distinguere i due usi; tuttavia questo è fortemente scoraggiato a livello stilistico).

I costruttori stessi sono chiamati come al solito con la keyword *new.* Se non viene dichiarato nessun costruttore, viene aggiunto quello di default; altrimenti, non viene proprio generato e quindi va creato manualmente assieme ad eventuali altri costruttori definiti dall'utente. Alcune librerie e strumenti richiedono espressamente la presenza di un costruttore di default: se viene dichiarato un costruttore esplicito, va quindi dichiarato anche uno privo di argomenti altrimenti questi strumenti non possono funzionare.

La gestione degli errori avviene con un sistema di *Tipi di Eccezione*, in cui le Eccezioni sono oggetti, ma vengono create ed usate in particolari situazioni, in particolare per le pratiche della programmazione Funzionale che tende ad evitarle o confinarle al minor uso possibile. Non vengono più molto utilizzate e tuttora il loro utilizzo è fonte di discussione. Tutte le eccezioni derivano dalla classe *Throwable*, in cui vi sono due discendenti principali, in particolare *Exception*, errori nonostante i quali il programma prosegue ed *Error*, errori per cui il programma non può proseguire.

È ancora considerato tutto sommato pratica normale nella realizzazione di codice "di libreria" o comunque molto generico, ma generalmente evitato in codice applicativo e più specifico. Molto si gioca sulla definizione di "eccezionalità" delle possibili condizioni di errore. Sottoclasse particolare è la *RuntimeException*, quindi ogni errore recuperabile e non serve dichiararla, in quanto è lanciata direttamente dalla JVM; tipica istanza è la *nullptr exception*, variabile che non punta a nessuna istanza di oggetto.



Eccezioni derivate da *RuntimeException* ed *Error* sono dette *unchecked exceptions* e non necessitano dichiarazione nella definizione di un metodo; tutte le altre, discendenti da *Exception* o *Throwable* sono *checked exceptions* e devono essere dichiarate nella definizione di un metodo.

Si parla poi di classi interne, dette *nested classes*, le quali posseggono le classiche quattro visibilità e sono una categoria di classi che si comporta in modo analogo agli altri casi. In queste abbiamo le *static nested classes*, classe dichiarata come statica all'interno di un'altra classe.

Altre classi interessanti di questo tipo sono le *inner classes*, parte dello stato di un oggetto del tipo ospite e, quindi, hanno lo stesso ciclo di vita ed ha un riferimento privilegiato all'oggetto ospitante. Detta in maniera più semplice, sono semplicemente classi non statiche dentro altre classi e non possono dichiarare membri statici ma solo membri di istanza. Le classi *static nested* sono spesso legate a qualche design pattern e solitamente vengono messe al di fuori della classe ospitante, per delineare meglio il legame esistente.
Le inner classes vengono invece utilizzate come meccanismo di sicurezza e sono da usare con cautela, in quanto può essere complesso il loro utilizzo.

Esempio di static nested class:

```
package it.unipd.app2020;

public class App {
  class Bar {
    int a;
  }

  String s;
}

App a = new App();
App.Bar b = a.new Bar();
```

App.Foo non ha un accesso privilegiato alla variabile s: la chiama App.s come ogni altra classe del package.

Esempio di inner class:

```
package it.unipd.app2020;

public class App {
  static class Foo {
    int a;
  }

  static String s;
}
```

In questo caso, un oggetto di App.Bar non può esistere (non può essere creato) senza che ci sia un oggetto di App che lo contiene.

Come blocchi di codice anonimi abbiamo gli *inizializzatori*, eseguiti in sede di inizializzazione di classe o di oggetto e sono blocchi di codice anonimi. Possono essere *static*, eseguiti quindi lessicalmente al caricamento della classe stessa. L'uso dei blocchi statici non è comune in quanto potrebbe essere pesante per il caricamento del programma. I blocchi di inizializzazione privi di indicazioni sono eseguiti lessicalmente durante la creazione di ciascuna istanza di oggetto della classe. In particolare, sono eseguiti dopo il supercostruttore (costruttore della superclasse) ma prima di qualsiasi altro costruttore.

```
package it.unipd.app2020;

public class App {

  String s = "foo";
  int l, j;

  App(int param) {
    j = param;
  }

  { l= s.length(); }
}
```

Scritto da Gabriel

In generale dato che sono anonimi meglio evitarne o quantomeno ridurne l'utilizzo; possono dar luogo a situazioni strane ed errori difficili da scovare.
Da quanto sembra, i blocchi anonimi possono anche essere usati, all'interno di una classe, per creare un tipo atto solo ad un contesto (quindi il metodo anonimo funge da tipo per un singolo oggetto).
Da qui deriva l'effettiva ambiguità nell'utilizzo dei blocchi anonimi.

Java può ereditare esclusivamente da una sola superclasse, ereditandone codice e parte dello stato. Una sottoclasse ai membri pubblici, a quelli protetti e ai package, non ai membri private. Ciò evita il problema *dell'ereditarietà a diamante*, la selezione di un metodo che arriva da molteplici percorsi di ereditarietà (l'introduzione dei metodi di default nelle interfacce ha fatto tornare questo problema).

```
package it.unipd.pdp2021;

class App {

  private int a;
  protected int b;

}

class Foo extends App {
  private String c;
}
```

Una sottoclasse è anche un sottotipo della classe che estende, quindi usata quando richiesto dalla superclasse. Tutti i metodi di Java sono *virtual*, quindi tutti i metodi possibili all'esecuzione si scoprono a runtime.
(Qui Foo ha visibilità su App::b, ma non su App::a).

Ricordiamo che le sottoclassi si dichiarano con la keyword *extends*.
Le classi invece dichiarate come *final* non possono essere usate come superclassi, quindi non ne deriviamo sottoclassi.

```
package it.unipd.pdp2021;

abstract class App {
  private int a;
  protected int b;
}

class Foo extends App {
  private String c;
}

App app = new App(); // Errore di compilazione
App foo = new Foo(); // OK
```

Contrariamente a questo, esiste la keyword *abstract* che dichiara che una certa classe debba essere usata come superclasse, quindi non istanziabile direttamente; ciò darebbe errore di compilazione.

Abbiamo anche le classi dichiarate *sealed*, elencando i possibili sottotipi permessi (con la apposita keyword *permits*).

```
public abstract sealed class Shape
  permits com.example.polar.Circle,
          com.example.quad.Rectangle,
          com.example.quad.simple.Square { }
```

Introdotta come preview in Java 15, ulteriore preview in 16 e come feature in 17, questa funzionalità è in realtà presente come supporto alla futura gestione del Pattern Matching, per permettere un ricco controllo di esaustività delle alternative. Estende la semantica di *final* ad un caso intermedio fra la possibilità generica di ereditare dalla classe e la sua impossibilità.

Sebbene classicamente l'ereditarietà nasca come metodo principe per il riuso del codice e per l'organizzazione dei tipi nell'OOP, già il GOF (Gang of Four dei Design Patterns) evidenziava i limiti di questo approccio suggerendo maggiore enfasi sull'uso della composizione in tutti i casi in cui ciò sia possibile. All'ereditarietà vengono lasciate solo quelle casistiche che le competono più strettamente.

Il codice di dominio dovrebbe avere alberi di ereditarietà bassi, se possibile, dando maggior enfasi alla composizione in tutti i casi possibili. Implicitamente tutte le classi derivano da *java.lang.Object*, ereditandone alcuni metodi fondamentali, come *hashCode*, riconoscendo oggetti diversi tra di loro tramite una computazione hash, rilevata sulla memoria effettiva, *equals*, riconoscendo l'uguaglianza tra tipi di oggetti, *toString*, dando l'emissione di un oggetto visto come stringa quando viene implementato a console. Vedremo come questi metodi siano fondamentali nella maggior parte delle API. Molto spesso, risulta di grande importanza che siano correttamente reimplementati nella semantica precisa del tipo.

Scritto da Gabriel

Classi e tipi (2) - interfacce, annotazioni, array, tipi primitivi, record (lezione 3), Istruzioni (Inizio lezione 4)

In Java, l'unità principale di organizzazione del codice è la Classe, ma non è l'unica.
Il primo costrutto che andiamo ad analizzare è l'*interfaccia*, che dichiara le caratteristiche di un tipo ma senza fornire una precisa implementazione; le classi che la dichiarano la devono implementare per contratto. Un'interfaccia può essere estesa solo da un'altra interfaccia; ciascuna di queste può avere visibilità pubblica o di package e i membri di ciascuna interfaccia sono pubblici.

```
package it.unipd.app2020;

interface Baz {
  int TEST = 1;
  void bar();
  String desc(boolean b);
}

class Foo extends App implements Baz {
  private String c;
  void bar() {};
  String desc(boolean b) { return ""; }
}
```

La classe Foo è obbligata a fornire un'implementazione di *desc()*, a meno di non essere astratta.
Quando all'interno di una classe, è possibile modificarne il tipo di accesso; come al solito, è possibile contenga costanti, metodi astratti e statici. Implicitamente i metodi di un'interfaccia sono di tipo *abstract final*. Da Java 8 in poi, *un'interfaccia può contenere anche metodi di default* (essi reintroducono il Diamond Problem) oppure metodi privati. Il loro uso principale è di stabilire un contratto tra le implementazioni e l'utilizzatore.

È comune, per esempio, che una libreria fornisca un insieme di interfacce per affrontare un certo problema, ed un insieme più ampio di implementazioni con caratteristiche differenti, che possono essere selezionate a runtime a seconda delle circostanze. Il codice utilizzatore non dipende dalla implementazione, ma può usare quella che gli viene fornita. L'asimmetria con l'ereditarietà di classi consente di risolvere le ambiguità.

Una interfaccia può essere estesa solo da un'altra interfaccia e può avere visibilità pubblica o di package. Cercare di indicare altrimenti non viene accettato dal compilatore. Per completezza, una interfaccia può essere static, private o protected quando è definita all'interno di un'altra classe. Una interfaccia può contenere dichiarazioni di costanti, metodi astratti e metodi statici. ogni variabile dichiarata in una interfaccia è implicitamente public static final, vale a dire costante.

Non è necessario indicare un metodo come abstract. Tutti i metodi sono implicitamente public. Una interfaccia può contenere anche la definizione di interfacce interne. Da Java 8 in poi, le interfacce contengono metodi di default o metodi privati (questi ultimi essendo tali non causano Diamond Problem).

Nella realizzazione di Java 8, gli autori di Java si sono trovati a dover affrontare un problema molto complesso:

- da un lato, la libreria standard necessitava di un profondo aggiornamento, sia per correggere errori storici che per includere innovazioni stilistiche e cambiamenti nel modo di programmare che si erano accumulati in oltre 15 anni
- dall'altro, per le caratteristiche delle Interfacce e del linking dinamico di Java, non era possibile modificare una interfaccia senza "rompere" tutto il codice che la utilizzava. Nel caso della libreria standard, questo era un problema enorme.

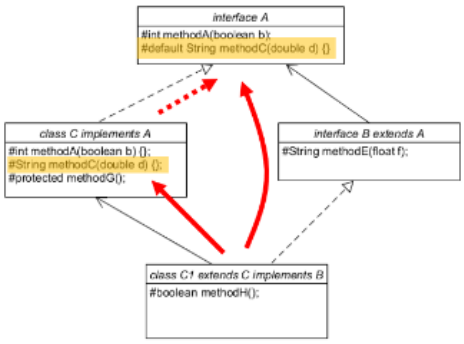
Aggiungere un metodo ad una interfaccia richiedeva di modificare tutte le implementazioni, aggiungendo a ciascuna il relativo corpo del nuovo metodo. La soluzione è diventata parte della *JSR335* che introduce il concetto di *default method* (prefissato dalla keyword default e con una serie di regole indicate per poter risolvere la compatibilità tra codice scritto prima e dopo la modifica dell'interfaccia). Alle interfacce viene permesso di avere dei default method, cioè dei metodi implementati che si comportano in modo simile a quello delle superclassi. Questo di fatto re-introduce l'ereditarietà multipla in Java.

I metodi di default hanno un *body* e forniscono delle funzionalità aggiuntive alle classi e le classi implementanti devono esplicitamente specificare quale metodo di default è usato e quale sovrascritto (override).

```
interface Top {
  default String name() { return "unnamed"; }
}

interface Left extends Top {
  default String name() { return getClass().getName(); }
}

interface Right extends Top {}

interface Bottom extends Left, Right {}
```

In questo caso, Bottom usa l'implementazione di Left in quanto più vicina in senso ereditario. Le regole per le interfacce sono separate da quelle delle classi.

Per mezzo dei default method una interfaccia può essere modificata con nuovi metodi senza che le implementazioni debbano essere toccate; se il metodo nuovo non è gestito dalla classe, viene usato quanto dichiarato nell'interfaccia. Tuttavia, il Diamond Problem si ripresenta (come si vede a lato):



Il Diamond Problem viene rilevato al momento della compilazione: se la gerarchia di ereditarietà ed implementazioni di una classe porta ad una ambiguità nella selezione di un metodo, il compilatore segnala un errore, che può essere risolto solo modificando il codice.
La gerarchia dei tipi deve essere rivista per conservare uno solo dei percorsi che portano al metodo ora ambiguo. Questo può costituire un problema molto laborioso da risolvere. Quello che si osserva, nella pratica, è che più il codice è specifico, più bassa dev'essere mantenuta la gerarchia delle classi; solo codice particolarmente generico, e accuratamente studiato, può permettersi una gerarchia molto profonda.

Altra cosa interessante, nel caso delle interfacce, il cui nome comincia con una chiocciola (@), dette *annotazioni*, applicate sintatticamente a classi, metodi e variabili e sono disponibili al momento dell'esecuzione. Arricchiscono le classi di *metadati,* consentendo la rilevazione durante l'uso e la compilazione rimanendo disponibili a runtime; grazie a questa cosa, a runtime è possibile avere una selezione delle classi per nome o per caratteristiche.

Annotazioni standard sono la *@Deprecated*, segnalando un metodo che verrà rimosso in futuro, oppure *@Override*, che segnala un membro che sostituisce o implementa un membro di superclasse o interfaccia. Se non implementata in quest'ultimo caso, darà errore.
Analogamente abbiamo *@SuppressWarnings*, abbastanza autoesplicativo e usato quando effettivamente serve logicamente oppure *@FunctionalInterface*, usata se possibile nelle lambda espressioni qualora detenga un singolo metodo di utilizzo.

```
class Foo extends Bar {
  @Override
  String methodB() { return ""; }

  @Deprecated
  @Override
  int methodOld { return 0; }

  @SuppressWarnings("unused")
  private boolean b = false;
}
```

Le annotazioni possono avere parametri e anche metodi; il compilatore può eseguire degli *Annotation Processors*, producendo nuovi file a partire da singole annotazioni. Il loro uso è analogo alle macro.
Prima di Java 8, l'altra rivoluzione in Java fu la versione 5 del 2004, introducendo i cosiddetti *Generics/1-kind parametric types*.
Una classe o interfaccia/metodo viene definita *generica* quando dichiara uno o più parametri di tipo che possono essere specificati in seguito.

Un esempio segue:

```
interface MappableList< T > {
  void add(T element);
  T head();
  List< T > tail();
  < M > List< M > map(Function< T, M > xform);
}
```

Una eccezione non può essere una classe generica. In una classe o interfaccia, il parametro di tipo va indicato dopo il nome. In un metodo, prima del tipo di ritorno: nell'esempio, la dichiarazione del metodo map indica il parametro di tipo M nella sua firma.

Il parametro *T* può essere usato come un tipo, specie se non si sa ancora che tipo sia. Qualora venga creata un'istanza della classe, implementazione dell'interfaccia o chiamata del metodo, è necessario specificare un tipo concreto al posto del parametro.

```
class StringList implements MappableList< String > {
  public void add(String element) {};
  public String head() { return ""; };
  public List< String > tail() {
    return Collections.emptyList();
  };
  public < M > List< M > map(Function< String, M > xform) {
    List< M > res = new ArrayList<>();
    for (String s : Arrays.asList(""))
      res.add(xform.apply(s));
    return res;
  };
}
```

Soprattutto con i metodi, e nelle versioni di Java più recenti, sono sempre di meno i casi in cui è realmente necessario specificare il tipo del parametro concreto; il compilatore è sempre più evoluto nel riconoscere il tipo desiderato dal contesto, quando possibile.

Si possono esprimere alcuni vincoli sui parametri per specificare meglio il loro tipo come nel caso seguente (dove l'interfaccia viene implementata solo specificando un tipo che implementi l'interfaccia Comparable):

```
interface SortableList< T extends Comparable < T > > {
  void add(T element);
  T head();
  List< T > tail();
}
```

Purtroppo, a causa di problematiche di compatibilità con il codice precedente, le informazioni sui tipi generici vengono cancellate al runtime e non sono più disponibili dopo la compilazione. Questo è stato un grosso cruccio per molte librerie e strumenti che cercavano di implementare algoritmi generalizzati su più tipi. Nonostante i loro limiti, i tipi generici permettono di realizzare in Java codice particolarmente espressivo in modo molto più semplice che, per esempio, con l'approccio dei template in C++.
L'uso più comune dei generici permette di evitare la duplicazione del codice e di semplificare l'uso delle classi contenitore; uso comune di questo costrutto è all'interno delle *collezioni*, quindi parliamo di liste, insiemi e code.

Non tutto in Java è oggetto, dato che il focus iniziale era rivolto alle piattaforme embedded; i tipi di dato fondamentali infatti non sono oggetti ma vengono detti tipi *primitivi*, quindi tutti i tipi *char, long, int*, etc.
Un valore primitivo non è un oggetto e non può essere indicato come parametro di tipo; ogni tipo primitivo ha quindi un corrispettivo Tipo non primitivo che può essere usato per trasportare lo stesso valore nelle situazioni in cui sia necessario.

Universo	Tipo Corto	Tipo Lungo
Interi	`Byte`, `Short`, `Integer`	`Long`
Decimali	`Float`	`Double`
Caratteri	`Char`	
Booleani	`Boolean`	

Universo	Tipo Corto	Tipo Lungo
Interi	`byte, short, int`	`long`
Decimali	`float`	`double`
Caratteri	`char`	
Booleani	`boolean`	

In Java 5 è stato introdotto il concetto di *autoboxing*, dato che il compilatore riconosce il contesto d'uso del valore primitivo, applicando la trasformazione necessaria; tuttavia può rivelarsi pesante da un punto di vista elaborativo, ma può anche dare problemi di tipo semantico (pesante in elaborazione l'autoboxing e portando a bug subdoli in fase di implementazione).

```
class Foo {
  int[] array;

  Foo() {
    array = new int[10];
    array[0] = 42;
  }
}
```

Gli *array*, invece, sono oggetti. La classe di un array discende direttamente da Object e viene creato dinamicamente come al solito nello heap. L'accesso ad un array è controllato a runtime, essendo la memoria di Java cosiddetta gestita.

```
enum Category {
  A, B, C, D;
}

Category cat = Category.A;
```

Un'altra categoria di classi utile è la *enum,* definito come tipo stringa che rappresenta numericamente un certo numero di elementi definiti alla dichiarazione. Essa non viene istanziata, ma è possibile usarne nella giusta sequenza i valori.
Formalmente, una classe E di questo tipo deriva da Enum<E>; se la enumerazione è una classe interna ed implicitamente statica e se non ha implementazioni specifiche, si considera *final*.

In Java, viene introdotta la *named tuple*, chiamata *Record* in Java, implementando dalla programmazione funzionale il concetto di tupla, introdotta a tutti gli effetti da Java 16 (2021).

La dichiarazione di un Record è molto semplice:

```
record Name(String firstName, String lastName){}
```

Nota importante: il record è *immutabile*.

Il record è *immutabile* e tutte le sue variabili sono considerabili *final* e non sono degli *Object*. Se dichiarati all'interno della classe, come sempre, saranno *static*. In automatico essi possiedono i membri privati con metodi di accesso pubblici, costruttore con tutti gli elementi del record e possiede metodi come *equals*, *hashCode, toString*.

Il costruttore così ottenuto è detto il costruttore canonico. Attenzione al fatto che i metodi di accesso hanno lo stesso nome dei membri del record, senza nessun prefisso get. Questo è contrario ad una consolidata (ma sempre meno sopportata) tradizione di Java, detta *Java Bean*.

Ai record mancano tuttavia metodi di modifica dello stato e non hanno concorrenza. È possibile definire metodi in un Record, reimplementare uno dei metodi generati automaticamente, dichiarare un record generico o implementare una interfaccia.
In questo modo, il Record diventa una scorciatoia per definire tutte le classi che normalmente modellano valori di dominio trasportati da un punto ad un altro del sistema o scorciatoie di creazione delle classi, avvicinandosi alla forma mentis della programmazione funzionale, per esempio implementando il *Pattern Matching* (cioè assegnare sotto forma di switch, quindi componendo più casi, un insieme di tipi sfruttando le lambda dei linguaggi funzionali).

Riassumiamo quindi i *modificatori di classe*, sempre prima alla parola chiave *class* e sono naturalmente keywords comprese nel linguaggio. Abbiamo:

- *abstract*, estesa da un'implementazione;
- *final*, non estesa da un'implementazione;
- *strictfp*, il codice della classe con operazione FP (floating point) restrittiva, implementando solo operazioni float/double.

Scritto da Gabriel

Riassumiamo quindi i *modificatori di metodo*, applicati appunto ad un metodo specifico. Abbiamo:

- *abstract*, estesa da un'implementazione di classe di estensione;
- *static*, legato alla classe e non ad un'istanza;
- *final*, non può essere reimplementato da una classe di estensione;
- *strictfp*, il codice della classe con operazione FP (floating point) restrittiva, implementando solo operazioni float/double;
- *native*, implementato da una libreria nativa;
- *synchronized,* il metodo può essere usato da un solo thread per volta.

Il codice in Java viene contenuto all'interno di blocchi delimitati da parentesi graffe, al cui interno stanno *statements* separati dal carattere ";". Non ci sta quindi garanzia che l'ordine di esecuzione delle istruzioni sia lo stesso del codice. Un primo tipo di istruzione è la *dichiarazione*, dando un nome e un tipo, sia nel caso di variabili, classi, ecc. In quel caso sono dette locali.
Solitamente hanno un tipo, nome ed eventuale valore di inizializzazione; similmente ad *auto* in C++, il compilatore cerca di capire e dedurre il tipo da solo grazie alla keyword *var*. Può esserci anche una lista vuota di parametri <>, anche questa sarà dedotta dal compilatore.

Un'istruzione può produrre valori e quindi è un'*espressione*, che produce un valore anche nel caso questo non venga usato (ad esempio quando chiamano un metodo senza preoccuparci del suo valore di ritorno). Nella grande maggioranza dei casi, le espressioni hanno un comportamento atteso.
Altri esempi di valori sono i *valori letterali*, ad esempio i caratteri, le stringhe, i booleani, ecc.
Nei casi di *byte* e *short,* non è prevista la dichiarazione di una costante; semplicemente ne si forza il cast verso il tipo stesso.

Le costanti di tipo String possono essere su più righe (*text block*), sono oggetti senza bisogno dell'operatore *new* e sono immutabili. La parola chiave *null* indica il valore nullo e il riferimento che non punta a nessun oggetto. Viene sempre convertito in ogni altro oggetto e non si può esprimere in termini di valore.
Consegue l'assegnamento "=", operatore che rappresenta un'istruzione di assegnazione dell'espressione ad una variabile specifica.

La chiamata di un metodo è un'espressione come le altre, ritornante un'espressione.
La creazione di un oggetto è per alcuni versi la chiamata di un metodo che ritorna il nuovo oggetto. Anche questa è un'espressione.
Una interfaccia può essere instanziata direttamente, fornendo l'implementazione al momento della creazione. Quindi (si noti che l'interfaccia è Comparator, parte dello standard):

```
Comparator< > reverse = new Comparator< String >() {

    @Override
    public int compare(String o1, String o2) {
      return -o1.compareTo(o2);
    }

  };
```

Un'interfaccia di questo tipo può essere considerato tipo anonimo; il loro utilizzo è stato rimpiazzato dalle lambda espressioni. Operatori vari usati sono "+", per concatenare stringhe, "[]", subscripting ed accesso all'elemento i-esimo dell'array e l'operatore ternario, un *if* in linea diciamo.
Le parole chiave *this* e *super* permettono di usare nel primo caso l'istanza corrente, utile nel caso di ambiguità di denominazione o per capire usare il significato di una espressione, oppure l'utilizzo di un oggetto padre della superclasse in ordine di gerarchia nel secondo caso.

Caso d'uso del *this*:

```
class Foo {
  public final int idx;
  public final String title;

  public Foo(int idx, String title) {
    this.idx = idx;
    this.title = title;
  }
}
```

Caso d'uso del *super*:

```
class A {
  public final int a;
  public A(int a) { this.a = a; }
}

class B extends A {
  public final int b;
  public B(int a, int b) {
    super(a);
    this.b = b;
  }
}
```

Lezione 4 – Istruzioni: condizioni, try/catch – Lezione 5 – Libreria standard: moduli

Una delle più grandi novità di Java 8 furono le *lambda expressions*, rendendo più semplici e compatte espressioni verbose. Vengono usate sia nelle pratiche funzionali che nelle SAM (interfacce con un metodo solo/Single Abstract Method).

La sintassi della **lambda expression** è la seguente:

```
( < lista parametri > ) -> istruzione
```

Il compilatore individua il tipo atteso e, se il tipo atteso corrisponde ad una SAM, implementa l'interfaccia come istruzione del metodo singolo. Java non diventa per questo un linguaggio funzionale.
Da sole, le seguenti espressioni non hanno tipo:

```
() -> 42
() -> { return 42; }
() -> { System.gc(); }
```

```
(int x) -> { return x+1; }
x -> x+1

(int x, int y) -> x+y
(x, y) -> x+y
```

Questo rappresenta un modo compatto per scrivere l'implementazione di una singola interfaccia:

```
import java.util.function.Function;

Function< Integer, String > f = x -> "%d".formatted(x);

Function< > g = new Function< Integer, String > {
  String apply(Integer x) {
    return "%d".formatted(x)
  }
}
```

La combinazione delle *lambda expression*, l'inferenza (quindi deduzione dal contesto delle espressioni) tramite la keyword *var* (quindi tipo *auto* in C++) e *diamond operator* (cioè < >) scrivono un codice conciso e comprensibile.

I costrutti condizionali in Java non sono espressioni, bensì istruzioni; ciò significa che eseguono blocchi di codice separati a seconda del valore della condizione. A tale scopo abbiamo i vari *If Then Else* o gli *switch case*. Da Java 14 la sintassi di switch è un'espressione, in grado quindi di ritornare un valore. Pur essendo simile esteriormente, vi sono delle differenze. Vi sono due sintassi disponibili.

Un'espressione dove viene assegnato il valore (distinguendo anche al posto di *break*, la keyword *yield, in* cui viene "data precedenza" (da cui yield) all'assegnazione del valore nell'espressione switch di cui fa parte prima di chiudere il blocco):

```
String weekPart= switch (day) {
  case LUN:
    System.out.println("Inizio");
    yield "Inizio settimana";
  case SAT, DOM: {
    System.out.println("Fine");
    yield "Weekend!";
  }
  default:
    yield "Nel mezzo...";
}
```

Ogni case per ritornare un valore deve contenere *yield* (non ci sta *fallthrough,* cioè dover forzosamente saltare ad una successiva istruzione *yield*) e l'elenco dei casi deve essere *esaustivo* (quindi deve esserci almeno un caso di default o comunque la lista di tutti i casi).
Anche le istruzioni di iterazione sono ispirate dal C; naturalmente parliamo del caro vecchio *while,* o anche la sintassi *do while*.

Un appunto (inclusione del bit del segno con lo shift, lavorando in bitmask):

```
i >>>= 4;
```

Si ha inoltre il *for*, usata anche per un ciclo su un oggetto che implementa l'interfaccia *Iterable* oppure un array. Cioè (considerando che l'ordine di iterazione viene dato dal contenitore):

```
int[] vals = new int[] { 5, 4, 3, 2, 1 };
for ( int i: vals ) System.out.println(i);
```

In questo modo noi stiamo dicendo al compilatore di scorrere tutti gli elementi di una struttura; ciò potrebbe permettere al compilatore di effettuare certe ottimizzazioni.

Seguono keywords come *break*/*continue*, interrompendo il ciclo di iterazione oppure proseguirla. Ogni istruzione può essere preceduta da un'etichetta, per saltare ad un punto preciso (*goto*).
Similmente un'altra istruzione è *return*, concludendo la chiamata al metodo attuale e ritornando il controllo al chiamante.

I metodi possono lanciare eccezioni e devono essere gestiti tramite un blocco *try/catch*.
Un esempio molto semplice:

```
class FooException extends Exception {}

class Foo {
  void a() throws FooException, BarException { return; }
  void b() throws BarException {
    try {
      a();
    } catch (FooException e) {
      e.printStackTrace();
    } finally {
      System.out.println("Always");
    }
  }
}
```

Nel blocco *finally* vi sono le istruzioni sempre eseguite a chiusura del codice, prima di ritornare il controllo al chiamante oppure dopo (per gestire normalmente risorse dell'esecuzione), a differenza del *catch* per esempio, che viene eventualmente eseguito; essi seguono un ordine lessicale, a seconda dell'importanza di ciascuna (es. IOException rispetto a FileNotFoundException, meglio mettere la prima in precedenza alla seconda, in quanto la seconda discende dalla prima).

Un'altra forma dell'istruzione *try* è detta *try-with-resources*, dichiarando una serie di risorse (quindi di oggetti) che devono essere chiuse alla fine dell'esecuzione del programma. Devono implementare l'interfaccia *AutoClosable*, dato che verranno certamente chiuse. In questa forma *catch/finally* sono opzionali. Esempio:

```
static String readFirstLine(String path)
    throws IOException {
  try (BufferedReader br =
      new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

Per lanciare esplicitamente eccezioni, abbiamo l'istruzione *throw*, richiedente un oggetto discendente da *Exception* e lanciato come fosse errore. Esempio:

```
class FooException extends Exception {}

class Foo {
  void a() throws FooException, BarException {
    throw new FooException();
  }
  void b() throws BarException {
    try { a();
    } catch (FooException e) { e.printStackTrace();
    } finally {
      System.out.println("Always");
    }
  }
}
```

Esiste anche il blocco vuoto oppure anche istruzioni vuote; alcuni IDE lo segnalano come errore, altri come warning, tuttavia sintatticamente è considerato corretto.

Vi è anche la keyword *assert*, che verifica determinate condizioni al momento dell'esecuzione del programma. Se l'espressione da verificare ritorna *false*, viene lanciato un errore. Queste possono anche essere totalmente ignorate, se richiesto. Esempio:

```
assert !importantList.isEmpty();
```

I JavaDoc sono le documentazioni dei metodi e classi e sono organizzati per "moduli", introduzione di Java 9. Un *modulo* controlla cosa è accessibile all'esterno dello stesso e in generale viene definito come *insieme di package e di tipi*. Il loro uso (da cui ispira il nome) era la modularizzazione, quindi la separazione del JDK in parti più piccole creando apposite distribuzione con i moduli necessari. Il progetto è riuscito solo in parte, in quanto i moduli servono solo in situazioni apposite e complessivamente si sono diffuse altre soluzioni (container, ecc.).

Parliamo subito della gestione I/O contenuto nel package *java.io,* le cui principali astrazioni sono *File, InputStream* ed *OutputStream*. Usi comuni sono attraverso *Reader* oppure *Writer*, fornendo metodi semplici per lettura/scrittura di file testuali. Esempio:

```
BufferedReader rd = new BufferedReader(
    new FileReader(".hgignore"));
String line = rd.readLine();
while (line != null) {
  System.out.println(line);
}
rd.close();
```

La libreria standard è organizzata per gerarchia di capacità, promuovendo l'uso della composizione per creare le necessarie catene elaborative. Questa versatilità produce API prolisse/ingombranti, dando spazio a tutti i punti di accesso per i vari casi d'uso. L'oggetto System (parte di Java.Lang.System) possiede gli stream classici (*System.in, System.out, System.err).* Similmente, Java possiede *java.nio*, aggiungendo nuove astrazioni, gestione asincrona di I/O e miglioramenti di performance.

Altra parte vasta ed importante è la *Collections API,* quindi la API delle collezioni. L'interfaccia *Collection* è la radice della libreria. Essa contiene i metodi più generali (dimensioni, test di contenitore, aggiunta/rimozione), interfaccia specializzata da altre volendo, dato che non ci sono implementazioni dirette. Molti metodi dell'interfaccia sono marcati come "opzionali", quindi le singole implementazioni ci sono se effettivamente esistenti, altrimenti ritorneranno una *UnsupportedOperationException* se non supportano una certa operazione. Caso tipico: viste non modificabili di altre collezioni, che non permettono la modifica del loro contenuto.

Moltissime collezioni si aspettano un uso coerente del metodo *java.lang.Object#equals()* in quanto non esiste un vero uso del confronto fra oggetti con l'operatore == (si può pensarlo come confronto tra puntatori, ritorna vero solo se puntano alla stessa cosa). Esempio utile:

```
class Point {
  public int x, y;

  Point(int x, int y) {
    this.x = x; this.y = y;
  }

  Point twoTimes() {
    x *= 2; y *= 2;
    return this;
  }
}
```

```
Point a = new Point(2, 1), b = new Point(3, 4),
  c = new Point(2, 1);
a == b // false
a == a // true
a == c // false
a == a.twoTimes() // true
```

Altro esempio con *equals* e anche *hashCode*:

```
@Override
public boolean equals(Object other) {
  if (other instanceof Point) {
    Point o = (Point) other;
    return this.x == o.x && this.y == o.y;
  } else
    return false;
}
@Override
public int hashCode() {
  return (x & 0xffff) << 16 + (y & 0xffff);
}
```

Il seguente caso darà *true* invocando *twoTimes:*

```
Point a = new Point(2, 1), b = new Point(3, 4),
  c = new Point(2, 1);
a == b // false
a == a // true
a == c // false
a.equals(c) // true
a.equals(a.twoTimes()) // ?
```

Collections possiede vari metodi di utilità, quindi:

`binarySearch`	Ricerca in una lista
`disjoint`	Verifica se disgiunte
`fill`	Riempie una collezione
`min, max`	Trova massimo e minimo
`reverse`	Inverte l'ordine
`shuffle`	Ordina in modo casuale
`sort`	Ordina la collezione

`checkedTTT`	Controllo al runtime del tipo
`emptyTTT`	Collezione vuota
`syncronizedTTT`	Collezione sincronizzata
`unmodifiableTTT`	Vista non modificabile

Altre interfacce importanti sono *Iterator*, elencando una collezione un elemento alla volta, oppure *Iterable*, fornendo un *Iterator* per essere attraversata oppure *Spliterator*, che attraversa in sequenza ogni singolo elemento di una collezione oppure di flussi di risorse in parallelo.

Gli elementi di Iterator più importanti sono:

`next`	Prossimo elemento
`hasNext`	Verso se ci sono altri elementi
`remove`	Rimuove l'elemento attuale
`forEachRemaining`	Consuma il resto della collezione

`forEach`	Applica ad ogni elemento
`iterator`	Fornisci un `Iterator`
`spliterator`	Fornisci uno `Spliterator`

Alcune note: *remove* è opzionale, *forEachRemaining* accelera il consumo della collezione.
L'interfaccia *List* è un elenco ordinato di elementi, indirizzabili per posizione; sono permessi elementi duplicati. Fornisce un iteratore apposito (*ListIterator*) capace di movimento bidirezionale.
Implementazioni utili:

`ArrayList`	Ridimensionabile, basata su array
`LinkedList`	Basata su nodi concatenati
`Vector`	Legacy, basato su array, sincrono

List fornisce anche un metodo *of* per creare rapidamente una lista a partire da un elenco di oggetti.

```
var list = List.of(1, 2, 3);
```

Un'altra interfaccia che definisce un insieme è *Set*, contenitore di oggetti senza ripetizioni non ordinato. Pessima idea mettere un elemento in un set e cambiarne il significato di *equals*, in quanto si potrebbe non trovarlo più sulla base di una specifica implementazione.
Alcuni tipi di Set:

`AbstractSet`	Scheletro di implementazione
`HashSet`	Basato su HashMap
`LinkedHashSet`	Ordinato in inserimento
`TreeSet`	Dotato di ordine interno
`EnumSet`	Specializzato per le `enum`

Ulteriori sono *SortedSet*, insieme con un ordine totale e *NavigableSet*, insieme ordinato muovendosi e sfruttando l'ordine, cercando direttamente l'elemento minore/maggiore di un elemento dato ed *EnumSet*, con delle bitmask che sono estremamente efficienti.

Lezione 5 - Libreria standard

Andiamo avanti con le interfacce, in particolare parliamo della *Dequeue*, usata sia come stack che come FIFO (essa è una Double Ended Queue) togliendo sia dall'inizio che dalla fine.
Esistono due implementazioni: *ArrayDequeue,* implementata come array oppure *LinkedList*, che può implementare sia la lista linkata singolarmente che la struttura Dequeue.
Essa possiede due set di metodi differenti, quindi:

- *(add, remove, get)* che può lanciare eccezione
- *(offer, poll, peek)* che può ritornare un valore speciale, scegliendo se usare try/catch oppure usare il valore che viene ritornato in quel momento. *Peek* prende il primo oggetto di una queue senza rimuoverlo, *poll* al contrario lo rimuove sempre dalla cima della queue, *offer* aggiunge un elemento nella queue se possibile farlo senza violare la capacity della stessa.

Altra interfaccia utile è la *Map*, che rappresenta un'interfaccia chiave/valore. Gli oggetti usati come chiavi devono essere corretti per *equals/hashCode* correttamente definita. Valgono le stesse cautele di mutamento di stato di una chiave.

Scritto da Gabriel

L'interfaccia mette a disposizione tre diverse viste sui contenuti:

- un elenco di *Entry*
- l'insieme delle chiavi
- l'elenco dei valori, insieme iterabile perché potrebbero esserci valori ripetuti

Non sono permesse chiavi non uniche. Le implementazioni permettono il valore *null* a discrezione del caso.
Alcuni esempi di implementazione:

Classe	Implementazione
`HashMap`	Base, chiavi distinte per `hashCode`
`TreeMap`	Chiavi ordinate
`Hashtable`	Implementazione storica, sincrona

Classe	Implementazione
`EnumMap`	Specifica per chiavi `enum`
`WeakHashMap`	Chiavi "deboli", non impediscono la GC
`IdentityHashMap`	Specifica basata sull'identità

Nel caso della *WeakHashMap* usa riferimenti deboli quindi, se un certo oggetto non viene riferito, il garbage collector lo dealloca autonomamente.
Uso pratico di questa particolare struttura: la cache, in particolare si pensi agli hit/miss.
Similmente a prima, esistono le *SortedMap* e *NavigableMap* partendo da un ordine totale sulle chiavi.
Similmente ai casi di List/Set, ecc. posso costruire rapidamente mappe con il metodo *of*, lavorando con generici/inferenza di tipo per capire le variabili usate:

```
var map = Map.of("A", 1, "B", 2, "C", 3);
```

Una parte importante dell'aggiornamento di Java 8 è stata l'introduzione del concetto di *Stream*, aggiunto sotto forma di metodo per realizzare le varie implementazioni (*stream()*)
Viene considerato sequenza di elementi, decidendo se iterare o meno sugli elementi, con una sequenza di elementi potenzialmente infinita. L'obiettivo è la descrizione dell'elaborazione e la sua ottimizzazione, duale all'obiettivo di una collezione.

Le operazioni vengono composte sequenzialmente, in una pipeline, finché non arrivo ad un'operazione terminale, che produce il risultato. Tutte le istruzioni di stream sono *lazy*, proprio perché vengono eseguite solo se necessarie per un'operazione terminale.
Il codice che implementa la pipeline può quindi prendere decisioni su quali operazioni realizzare e quali saltare. Le operazioni intermedie non devono modificare gli elementi dello stream e nella maggior parte dei casi non hanno uno stato interno. Essi possono essere costruiti sia da collezioni di partenza sia altri tipi di astrazioni (file, canali di comunicazione, generatori casuali).

Le operazioni intermedie sono divise tra *stateful* e *stateless*, variando a seconda dell'efficienza della pipeline. Descriviamo di seguito le operazioni *stateful* (il risultato dipende da altre operazioni intermedie che ne modificano lo stato durante l'esecuzione nella pipeline), seguendo poi le *stateless* (non ricavano alcuno stato durante l'esecuzione della pipeline):

Stateless	Significato
`filter`	Solo gli elementi che soddisfano un predicato
`drop/takeWhile`	Escludi/mantieni elementi finché vale un predicato
`map`	Trasforma ogni elemento
`peek`	Esegue un'operazione senza consumare l'elemento

Stateful	Significato
`distinct`	Elementi distinti
`concat`	Concatena due stream
`limit`	Tronca lo stream
`skip`	Salta l'inizio dello stream
`sorted`	Ritorna uno stream ordinato

Seguono istruzioni terminali/generatori:

Terminale	Significato
`all/any/noneMatch`	Vero se uno/tutti/nessuno gli elementi soddisfano il predicato
`collect`	Riduce lo stream ad un risultato
`findAny/First`	Ritorna un o il primo elemento

Terminale	Significato
`flatMap`	Trasforma ogni elemento in nuovi elementi
`forEach/Ordered`	Esegue un'operazione per ogni elemento
`min/max`	Minimo o massimo
`reduce`	Riduce lo stream con una operazione associativa

Generatore	Significato
`generate`	Produce uno stream a partire da un `Supplier`
`iterate`	Produce uno stream applicando una funzione a partire da un seme

Nota: *Supplier* è un interfaccia che solitamente implementa un metodo solo: potenzialmente può essere usato per stream o generatore di istanze all'infinito.
In generale quindi sono un'astrazione utile e consente di descrivere il significato dell'elaborazione, invece del metodo, una volta descritto il calcolo fatto; il compito di organizzare tutto il calcolo appartiene proprio allo stream.

L'uso del *null* porta una serie di conseguenze, perché si deve capire come utilizzarlo. Questa ambiguità porta a numerosi idiomi di *programmazione difensiva*, in cui si cerca di mantenere l'esecuzione nonostante possibili problemi (esempio spesso presente, la gestione delle NPE/Nullptr Exception e dei valori nulli, oppure anche asserzioni/precondizioni). Sicuramente è buona programmazione quest'ultima sotto alcune circostanze, ma non in tutte; volendo si usa il concetto di *programmazione offensiva*, che conosce l'errore e cerca di agire controbilanciandolo e prevenendolo.

Diciamo che noi introduciamo un concetto che è una via di mezzo, cioè la classe *Optional*, definita in altri linguaggi come *Option* o *MayBe*, indicando che la classe rende esplicita la rappresentazione di un valore che potrebbe esserci oppure no. In questi casi meglio inizializzare direttamente il valore risolvendo le ambiguità; se ben implementato può evitare le NPE.

Altro problema interessante è la *gestione del tempo*, che comporta varie irregolarità in merito alla sua gestione (es. alcuni hanno usato 62 secondi al posto di 60, oppure la data del 30 giugno in merito all'aggiunta dei secondi bisestili, ecc.).
La prima API temporale di Java ruota attorno a *java.util.Date*, assorbita poi nel package *java.time*. Seguono inoltre *Instant*, singolo ed astratto istante nel tempo, nonché *LocalDate, LocalTime* e *LocalDateTime*, che rappresentano una data/ora/istante. Segue anche *ZonedDateTime* con l'informazione del fuso orario.

Ora finalmente facciamo degli esempi e parliamo del *TDD (Test Driven Development)*. Le fasi sono:

- pensare al problema e a come scrivere il test
- scrivere il test
- osservarlo fallire
- si scrive il codice più semplice possibile per farlo passare

Abbiamo un primo esempio con Fibonacci.
Altra idea che viene dai kata del karate descrive il fatto di cercare ripetutamente di replicare l'approccio di risoluzione del problema piuttosto che concentrarsi sul risultato del problema stesso.
Un altro esempio è il calcolo del punteggio del Bowling, considerando tiri vuoti, spare o strike.

Lezione 6/7 – Esempi svolti pt.1/2: Fibonacci/Bowling

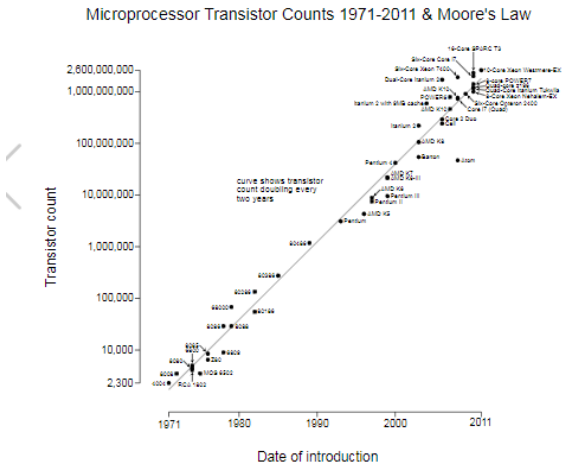
In questo esempio si ha una classe BowlingGame dove noi implementiamo il codice e dobbiamo produrre un metodo *score()* che calcoli il punteggio della partita.
Abbiamo una serie una serie di classi di test. In questo caso abbiamo diverse variabili da considerare:

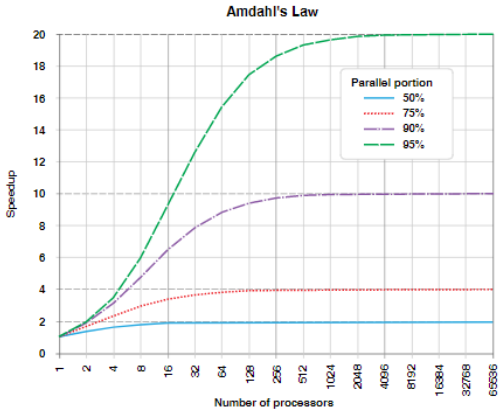
- il numero di tiri, 2 per tutta la partita (9 frame), 3 nell'ultimo frame
- se in un frame il giocatore abbatte meno di dieci birilli, ottiene come punteggio il numero di birilli abbattuti (viene indicato con segno "-" e semplicemente indica che in un frame, si ha un solo tiro, l'altro non si conta)
- lo spare, quindi abbattendo dieci birilli in due tiri si conta come bonus il numero di birilli abbattutti con il tiro seguente
- lo strike, abbattendo dieci birilli con un solo tiro, si conta come bonus il numero di birilli abbattuti nei due tiri seguenti
- nel decimo frame, se si ottiene uno strike/spare nei primi due tiri, si ha un terzo tiro con cui calcolare il bonus per il punto segnato

Lezione 8 - Programmazione concorrente

Si intende con *programmazione concorrente* quindi la teoria e l'insieme di tecniche utili per la gestione di più processi contemporaneamente sulla stessa macchina.
Nelle macchine con una sola CPU essa è diventata collo di bottiglia/bottleneck, partendo infatti dalla macchina di Von Neumann, che però eseguiva una sola istruzione alla volta. Per questo diventa rapidamente molto poco efficiente.
Presto vengono introdotti dei channels nei mainframe IBM, rendendo "intelligenti" le periferiche leggendo in nastri e stampando risultati. Il modello economico affronta il problema del coordinamento facendo in modo esso diventi più efficiente.

La concorrenza diventa quindi problema del sistema operativo, che si trova a gestire molteplici attività contemporaneamente o in rapida successione. Dunque, le risorse disponibili vanno distribuite tra le attività.
La CPU è quindi efficace per un tempo maggiore, piuttosto che attendere l'esecuzione delle singole istruzioni. La legge di Moore (1965) fornisce risorse sempre crescenti, raddoppiando il numero di transistor a disposizione. Di fatto, in dimensioni così ridotte, gli effetti termici sono molto rilevanti e si cerca di risolvere queste problematiche moltiplicando i core, predicendo dove le prossime istruzioni possono saltare (caso Spectre/Meltdown).

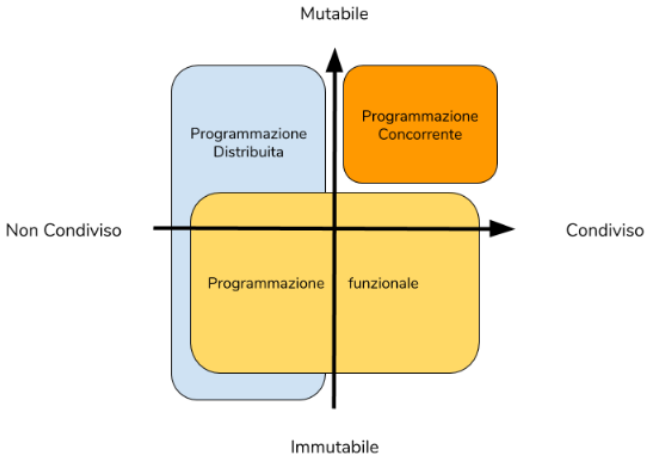




Tutta questa gestione diventa complicata ed insicura. La legge di Amhdal (1965) individua i limiti matematici all'efficienza della parallelizzazione.
Il parallelismo quindi usa più risorse per ottenere i risultati pià velocemente; nella realtà teniamo solo quelli più utili.

Parallelizzare le attività non è sempre possibile e il lavoro deve essere valutato in base alla parallelizzazione ottenibile e l'accelerazione ricavabile. Concettualmente in UNIX descriviamo un *processo* come un programma in esecuzione e tutte le sue risorse assegnate (memoria, canali I/O, interrupt/segnali, stato CPU). I processi non condividono risorse di norma, interagendo come entità separate, garantendo un uso efficace e paritario delle risorse. Per la simmetria/velocità della CPU nel resto del computer è quindi uno spreco. Il contesto della CPU deve essere salvato e messo da parte per evitare sprechi.

Per gestire quindi più linee di esecuzione all'interno dello stesso processo nasce il concetto di *thread,* che sono linee di esecuzioni concorrenti in uno stesso processo, risparmiando le risorse in caso di cambio di contesto. Di fatto, però, è in carico all'applicazione il problema della gestione e dell'accesso contemporaneo. Ciò è visibile dall'immagine a lato, con nodi che non condividono risorse fisicamente separati.



Usiamo quindi una serie di tecniche e strumenti per navigare in questa situazione. La *programmazione distribuita* implica la comunicazione fra entità che non possono avere stato condiviso, mentre la *programmazione funzionale* tratta principalmente dati immutabili, con stato che può essere condiviso.

La *programmazione concorrente*, quindi, unisce le due cose.
La *programmazione ad oggetti* riguarda tutti e quattro gli stati sopra disegnati.
Parliamo poi del *non determinismo*, sapendo che un'esecuzione concorrente è inerentemente non deterministica e potrebbe morire di fame (*starvation*), competizione delle stesse risorse (*race conditions*), nessuno dei due può proseguire perché uno dei thread attende la risorsa dell'altro.
Le linee di esecuzione scambiano continuativamente le risorse dando "l'impressione" di stare avanzando.

Nel caso di *deadlock* tuttavia, si hanno le *condizioni di Coffman*:

- Mutua esclusione → Risolvibile tramite algoritmi lock-free
- Attesa della risorsa e trattenimento della stessa → Risolvibile tramite assegnazione delle risorse in modo transazionale
- No preemption → Risolvibile tramite assegnazione delle risorse in modo transazionale
- Attesa circolare → Risolvibile tramite ordinamento dell'acquisizione

Queste sono necessarie perché avvenga un deadlock. Rimuovere la mutua esclusione può quindi non essere fattibile per certe risorse, richiedendo algoritmi specifici detti *lock-free* o *wait-free.*
Rimuovere l'attesa, quindi, può portare a situazioni di starvation o attese indefinite.
Introdurre la pre-emption può essere costoso ma anche impossibile; con l'uso degli algoritmi detti può essere introdotto un controllo cosiddetto *optimistic concurrency control* (spesso fatto nei sistemi DBMS attraverso le transazioni, registrando con un timestamp l'avvenuta transazione, validando le modifiche e rendendo le operazioni di commit e validazione atomiche).
Rimuovere la *circolarità* richiede di imporre un ordinamento alle risorse; ciò crea difficoltà, perché la linea di esecuzione è cosciente dell'esistenza delle altre e non sempre è facile da individuare/creare.

Seguono le tipologie di concorrenza:

Tipo	Strutture
Collaborativa	Co-Routines
Pre-Emptive	Processi, Threads
Real-Time	Processi, Threads
Event Driven/Async	Future, Events, Streams

Il concetto di *coroutine* fa riferimento a costrutti particolari che, volendo, possono essere importati e usati in Java puro.
Inoltre, si aspettano, in quanto *thread* più leggeri, una elaborazione meno pesante da un punto di vista computazionale e che sfruttano un pattern di codice ben definito (es. produttori/consumatori).
Maggiori al link: https://medium.com/@esocogmbh/coroutines-in-pure-java-65661a379c85

La descrizione completa delle quattro segue:

- concorrenza *collaborativa*, i programmi cedono il controllo ad intervalli regolari. Questo modello risulta essere rilevante in vari ambiti (embedded, very high performance, ecc.)
- concorrenza *pre-emptive*, il sistema operativo è in grado di interrompere l'esecuzione di un programma e sottrargli il controllo delle risorse per affidarle al programma seguente, avendo dei programmi privilegiati.
- concorrenza *real-time*, il sistema operativo deve garantire performance precise e numericamente specificate, per esempio gli interrupt entro un certo numero di ms (esempio utile che ho trovato io: FreeRTOS, struttura a superloop e permette l'esecuzione di un certo numero di task concorrentemente e precisamente, sulla base dei tick in ms).
- concorrenza *event driven/async*, dove i programmi dichiarano le operazioni che vanno eseguite e lasciano la decisione all'ambiente di esecuzione, decidendo quando e come assegnare le risorse. Quest'ultima sta diventando popolare nell'organizzazione delle applicazioni.

Parliamo quindi di *thread* in Java, i quali usano un oggetto di tipo Runnable nella costruzione:

```
/**
* Allocates a new Thread object.
*
* @param target the object whose run method
* is invoked when this thread is started.
* If null, this classes run method does nothing.
*/
public Thread(Runnable target)
```

Usando il metodo *start()* la linea di esecuzione procede immediatamente avviando l'attività del thread, avviando un nuovo percorso di esecuzione nella JVM, condividendo lo stesso heap (quindi chiamando quel metodo il percorso di esecuzione non è più univoco: da un lato, il metodo ritorna ed il programma chiamante prosegue, dall'altro il metodo chiamato viene eseguito contemporaneamente in una nuova linea di esecuzione).

```
/**
* Causes this thread to begin execution; the Java Virtual
* Machine calls the run method of this thread.
*
*/
void start()
```

Altro metodo che spesso useremo sarà *sleep()*, che mette in pausa il thread corrente per un certo periodo di tempo, specificando i millisecondi:

```
/**
* Causes the currently executing thread to sleep
* (temporarily cease execution) for the specified
* number of milliseconds, subject to the precision
* and accuracy of system timers and schedulers.
*
*/
static void sleep(long millis)
```

```
/**
* The Runnable interface should be implemented by any
* class whose instances are intended to be executed
* by a thread.
*/
@FunctionalInterface
public interface Runnable {

  /**
  * The general contract of the method run is that
  * it may take any action whatsoever.
  */
  void run();
}
```

Abbiamo poi l'interfaccia *Runnable*, che modella un compito da cui non ci si aspetta un risultato. In essa non è concesso lanciare eccezioni ed è definita come SAM.

Un esempio possibile è l'interfaccia *ThreadSupplier*, fornitore di thread che aspettano del tempo.
In questo codice, la lambda viene riconosciuta come implementazione di Runnable e viene usata come strategia di generazione dei tempi d'attesa; i metodi statici sui Thread controllano il comportamento del thread corrente.

```
@Override
public Thread get() {
  return new Thread(() -> {
    String name = Thread.currentThread().getName();
    long time = waitTime.get();
    try {
      Thread.sleep(time);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  });
  }
```

Seguono altri esempi con:

- uso di un singolo thread (il programma non termina dopo l'esecuzione del main(), ma attende che il thread completi la sua esecuzione; se usassimo un *import static* su *System.out*, accorceremmo le istruzioni di stampa):

```
public static void main(String[] args) {
  Thread a = new ThreadSupplier().get();

  out.println("Starting Single Thread");
  a.start();
  out.println("Done starting.");
}
```

- uso di molteplici thread lanciati in successione (usando *ThreadSupplier* come generatore di uno stream, ottenendoli in successione)

```
public static void main(String[] args) {
  var threads = Stream.generate(new ThreadSupplier());

  out.println("Starting Threads");
  threads.limit(10).forEach((Thread a) -> a.start());
  out.println("Done starting.");
}
```

Lezione 9 - Threads

Cominciamo ad approfondire l'uso dei thread esaminandone avvio e stato; diamo qui sotto la forma completa di thread specificando anche gli argomenti di solito opzionali (esempio il parametro *group* che può essere soggetto a restrizioni di sicurezza, a discrezione della JVM):

```
/**
* Allocates a new Thread object so that it has target as
* its run object, has the specified name as its name, and
* belongs to the thread group referred to by group, and
* has the specified stack size.
*
*/
public Thread(ThreadGroup group,
  Runnable target,
  String name,
  long stackSize)
```

Similmente, come visto ieri, *start()* ritorna al chiamante e lancia la linea di esecuzione parallela (con il metodo *run*()), o metodi come *getName()*, prendendo il nome e rendendo più facile la gestione dei log.
Il metodo che qualifica se è vivo/ancora attivo il thread è il booleano *isAlive()*.
Poi il metodo *run()* esegue l'oggetto Runnable nella stessa istanza di esecuzione in quel momento e non ritorna nulla.

```
final Thread observer = new Thread(() -> {
  out.println("(Start) Target live: " + tgt.isAlive());
  for (int i = 0; i < 10; i++) {
    try {
      Thread.sleep(100L);
      out.println("Target live: " + tgt.isAlive());
    } catch (InterruptedException e) {
      out.println(" Observer Interrupted");
      e.printStackTrace();
    }
  }
  out.println("(End) Target live: " + tgt.isAlive());
});
```

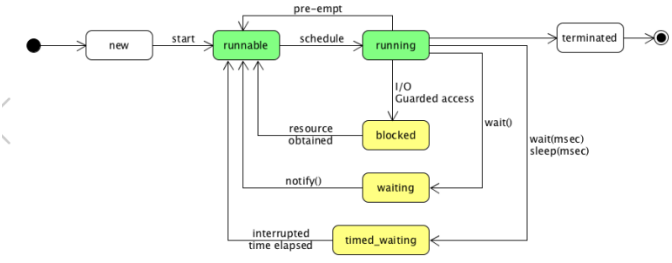
Altro metodo statico del thread in cui ci troviamo in quel momento si ha *currentThread()*. Il thread principale è il *main()*. Metodo di temporizzazione approssimato, dato dall'orologio dell'hardware, è lo *sleep()*.
Costruiamo un thread che osserva lo stato di un altro thread:
(lo fa per 10 volte ogni 100 ms)

```
public static void main(String[] args) {
  final Thread tgt = new ThreadSupplier(800L).get();

  // ...observer...

  observer.start();
  tgt.start();
}
```

Il thread osservato attende 800ms prima di uscire, quindi dovremmo notare il cambio di stato. Ancora osserviamo che la JVM non termina dopo l'avvio dei Thread.

Oltre ai classici stati (vivo/terminato) i thread hanno una serie di stati (come visibile dal seguente schema che segue il tipo enum). Essi sono:

- *new*, creazione del thread;
- *runnable*, a seguito del richiamo dello *start()*, mandando poi in esecuzione (volendo) il metodo *run();*
- *running*, una volta che ha ricevuto la risorsa e sta eseguendo istruzioni. Può uscire da questo stato se la CPU gli viene sottratta, salvando il suo stato di contesto o passa ad altro stato;
- *blocked,* se ha richiesto accesso ad una risorsa monitorata e aspetta la disponibilità dei dati;



java.lang.Thread.State

- *waiting,* il thread aspetta la disponibilità di alcuni dati, attendendo una risorsa protetta da un *lock()*.
- *timed_waiting,* attesa con un massimale di tempo dato dallo sleep, scaduto il quale torna Runnable
- *terminated*, mettendo il thread a disposizione della garbage collection, in quanto ha completato la propria esecuzione

Quando tutti i thread sono stati terminati, la JVM ritorna il controllo alla shell richiamata.

Parliamo quindi di *interruzioni* (fatte con il metodo *interrupt()*), lanciando una possibile eccezione oppure l'interrupt che può essere invocato anche da un altro thread.
Vediamo quindi un altro esempio, controllando se il thread è vivo e lo interrompe:

```
@Override public void run() {
  out.println("Target Thread alive: " + tgt.isAlive());
  for (int i = 0; i < 4; i++) {
    try {
      Thread.sleep(1000L);
      tgt.interrupt();
      out.println("Target interrupted.");
    } catch (InterruptedException e) {
      out.println("Interrupter Interrupted");
      e.printStackTrace();
    }
  }
  out.println("Target Thread alive: " + tgt.isAlive());
}
```

Nel main creiamo un ThreadSupplier e un interruttore; se interrompo un thread che non è vivo, non c'è nessun risultato:

```
public static void main(String[] args) {
  final Thread tgt = new ThreadSupplier(2000L).get();
  final Thread interrupter = new Thread(new Interrupter(tgt

  interrupter.start();
  tgt.start();
```

Possiamo impostare, per uno specifico thread, un gestore delle eccezioni che riceve quelle non intercettate (unhandled) e può quindi modificare la risposta del thread in seguito ad un comportamento non previsto.

```
/**
 * Set the handler invoked when this thread abruptly
 * terminates due to an uncaught exception.
 */
public void setUncaughtExceptionHandler(
    Thread.UncaughtExceptionHandler eh)
```

Creiamo uno specifico supplier per farci fornire dei threads che rilanciano l'eccezione di interruzione invece di gestirla. A tutti gli effetti, se interrotti, questi thread lanciano un'eccezione non gestita.

```
@Override
public Thread get() {
  return new Thread(() -> {
    String s = Thread.currentThread().getName();
    long t = waitTime.get();
    out.println(s + " will wait for " + t + " ms.");
    try {
      Thread.sleep(t);
      out.println(s + " is done wating for " + t + " ms." );
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
  });
}
```

Impostiamo l'exception handler sul thread bersaglio: vedremo che l'handler viene richiamato e gestisce l'eccezione.

```
final Thread tgt=new RethrowingThreadBuilder(2000L).get();
tgt.setUncaughtExceptionHandler((Thread t, Throwable e) -> {
  out.println("Thread " + t.getName() +
    " has thrown:\n" + e.getClass() + ": " + e.getMessage());
  });

final Thread interrupter = new Thread(new Interrupter(tgt));
interrupter.start();
tgt.start();
```

Già maneggiare i singoli thread richiede un po' di sintassi e l'amministrazione si complica al crescere del numero degli oggetti. La soluzione è cedere parte del controllo al sistema, in cambio di maggiore semplicità ed efficienza.
Diamo quindi il controllo all'interfaccia *Executor,* prendendosi il carico su nostra indicazione come gestire lo scheduling (separando l'esecuzione del compito dalla sua implementazione):
Il metodo più usato è proprio *execute*(), con argomento un Runnable.

```
/**
 * An object that executes submitted Runnable tasks.
 * This interface provides a way of decoupling task submissior
 * from the mechanics of how each task will be run, including
 * details of thread use, scheduling, etc.
 *
 */
public interface Executor
```

```
Executor executor = Executors.newFixedThreadPool(4);

var threads = Stream.generate(new ThreadSupplier());
out.println("Scheduling runnables");
threads.limit(10).forEach((r) -> executor.execute(r));
out.println("Done scheduling.");
```

Nel seguente esempio la JVM rimane attiva: l'ExecutorService rimane in attesa di nuovi compiti da eseguire, anche se il metodo main() è concluso.

Similmente, i compiti accodati vengono eseguiti da un solo thread:

```
Executor executor = Executors.newSingleThreadExecutor();

var threads = Stream.generate(new ThreadSupplier());
out.println("Scheduling runnables");
threads.limit(10).forEach((r) -> executor.execute(r));
out.println("Done scheduling.");
```

Diamo quindi alcuni esempi di esecutori:

CachedThreadPool	Riusa thread già creati, ne crea nuovi se necessario	ScheduledThreadPool	Esegue i compiti con una temporizzazione
FixedThreadPool	Riusa un insieme di thread di dimensione fissa	SingleThreadExecutor	Usa un solo thread per tutti i compiti
ForkJoinPool	Punta ad usare tutti i processori disponibili. Specializzato per il framework di fork/join		

Quest'ultimo esecutore (fork) è ottimizzato per risolvere problemi secondo la logica *work stealing* (uso una serie di thread worker per risolvere un problema con un approccio *divide and conquer*, quindi risolvendo i sottoproblemi; ne può eseguire fino a 32767).

Finora abbiamo lavorato con attività che non danno effetti collaterali (oggetti *Runnable*); ora parliamo dell'interfaccia *Callable*, che definisce dei compiti che producono un risultato:

```
/**
 * A task that returns a result and may throw an exception.
 */
@FunctionalInterface
public interface Callable < V > {
  /**
   * Computes a result, or throws an exception if unable
   * to do so.
   *
   * @return computed result
   * @throws Exception - if unable to compute a result
   */
  V call() throws Exception;
}
```

```
/**
 * An Executor that provides methods to manage termination
 * and methods that can produce a Future for tracking
 * progress of one or more asynchronous tasks.
 *
 */
public interface ExecutorService
  extends Executor
```

Un Executor non esegue Callable; usiamo quindi *ExecutorService*, parametrizzato in base al tipo di Callable (interfaccia implementata da una serie di *Executor*):

Il calcolo che prima o poi ritornerà un valore dà un oggetto F*uture,* che rappresenta un calcolo che prima o poi ritornerà un valore; in questo caso viene parametrizzato in base al tipo.

```
/**
 * Submits a value-returning task for execution and
 * returns a Future representing the pending results
 * of the task.
 *
 * @param T - the type of the task's result
 * @param task - the task to submit
 * @return a Future representing pending completion
 * of the task
 *
 */
< T > Future< T > submit(Callable< T > task)
```

```
/**
* A Future represents the result of an asynchronous
* computation. Methods are provided to check if the
* computation is complete, to wait for its completion,
* and to retrieve the result of the computation.
*
*/
public interface Future< V >
```

In generale, un interfaccia di tipo *Future* controlla il risultato di una computazione asincrona e vengono forniti metodi per controllare se la computazione è completa o se si deve attendere per il suo completamento, recuperando il risultato.

Il metodo *get()* ritorna il risultato una volta finito il calcolo; quando ciò avviene, si usa il metodo *isDone()*.
Diamo un esempio di esecutore, implementando una serie di attività e verificando quante sono state completate per mezzo di una stampa:

```
ThreadPoolExecutor executor =
  (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
Supplier< Callable< Integer > > supplier =
  new FactorialBuilder();
List< Future< Integer > > futures =
  new ArrayList< Future< Integer > >();
```

```
for (int i = 0; i < 10; i++)
  futures.add(executor.submit(supplier.get()));
while (executor.getCompletedTaskCount() < futures.size()) {
  out.printf("Completed Tasks: %d: %s\n",
    executor.getCompletedTaskCount(),
    format(futures));
  TimeUnit.MILLISECONDS.sleep(50);
}
```

Si usa qui sotto *FactorialBuilder*, che crea una serie di attività date da *Supplier* numericamente equivalenti alla serie di Fibonacci e al fattoriale eseguito incrementalmente. Di seguito i task completati*:*
In generale, tramite la lista di *Callables*, otteniamo il risultato di un *Future* che ha terminato (non necessariamente il primo, ma probabilmente uno dei primi), ottenendo una lista di *Future* nel momento in cui tutti sono stati completati.

Lezione 10: Sincronizzazione

Posso eseguire in metodo chiamato *invokeAny*, che esegue i processi presenti e restituisce il risultato di almeno una che ha completato con successo; pertanto, non restituisce eccezione (infatti non tutti i Callable, in questo stato, hanno completato l'esecuzione).

```
/**
 * Executes the given tasks, returning the
 * result of one that has completed successfully
 * (i.e. without throwing an exception), if any do.
 *
 */
< T > T invokeAny(
  Collection < ? extends Callable< T > > tasks)
```

```
/**
 * Executes the given tasks, returning a list of Futures
 * holding their status and results when all complete.
 * Future.isDone() is true for each element of the
 * returned list.
 */
< T > List< Future< T > >
  invokeAll(Collection< ? extends Callable< T > > tasks)
```

Questa chiamata ritorna solo dopo il completamento di almeno uno dei Future presenti, sapendo che quando il metodo ritorna, tutti i Future hanno completato.

Nei due precedenti esempi, rimaniamo in attesa di una serie di Callable e vediamo se li abbiamo chiamati tutti. Normalmente un *ExecutorService* rimane sempre in attesa di compiti da eseguire, impedendo alla JVM di terminare. Per fermarla bisogna chiamare esplicitamente *shutdown()*.

```
ThreadPoolExecutor executor =
  (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
var supplier = new FactorialBuilder();
var callables = new ArrayList< Callable< Integer > >();
for (int i = 0; i < 10; i++)
  callables.add(supplier.get());

out.println("Scheduling computations");
var futures = executor.invokeAll(callables);
out.println("Done scheduling.");
```

```
ThreadPoolExecutor executor =
  (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
var supplier = new FactorialBuilder();
var callables = new ArrayList< Callable< Integer > >();
for (int i = 0; i < 10; i++)
  callables.add(supplier.get());

out.println("Scheduling computations");
var result = executor.invokeAny(callables);
out.println("Done invoking: " + result);
```

Si ha poi l'attesa di completamento dell'attività in esecuzione fino alla richiesta di shutdown.

```
/**
 * Blocks until all tasks have completed execution after a
 * shutdown request, or the timeout occurs, or the current
 * thread is interrupted, whichever happens first.
 *
 */
boolean awaitTermination(long timeout, TimeUnit unit)
```

Si capisce poi se l'ExecutorService abbia ancora attività in corso con *isTerminated()*, metodo booleano, e poi abbiamo il metodo *shutdownNow()*, che ritorna una List<Runnable> e cerca di fermare tutti i task in attesa.

Ora, come si manifestano concretamente i problemi di uno Stato Mutevole e Condiviso?
Prendiamo l'esempio di una semplice interfaccia contatore:

```
/**
* A simple interface to a counter.
*/
interface SimpleCounter {

  public void add();

  public int getState();
}
```

```
class UnsyncCounter implements SimpleCounter {
  private int state = 0;
  public void add() {
    int current = state;
    try {
      TimeUnit.MILLISECONDS.sleep(
        Math.round(Math.random() * 100));
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    state = current + 1;
  }
}
```

Sostituendo la seguente classe alla precedente nel metodo main il comportamento diventa corretto. La sezione critica, cioè il metodo in cui viene modificato lo stato, è ora protetta.

I vari metodi di accesso ai Thread non sono modellabili con una struttura dati; a volte abbiamo bisogno di controllare come i Thread attraversino sezioni di codice.
Abbiamo esempi come classi di incrementazione (*Incrementer*) e di esecuzione di contatori (*RunCounter*):

```
class Incrementer implements Callable< Boolean > {
  private SimpleCounter counter;
  Incrementer(SimpleCounter counter) {
    this.counter = counter;
  }
  @Override
  public Boolean call() {
    IntStream.range(0, 10).forEach((i) -> counter.add());
    return true;
  }
}
```

Eseguiti tutti i thread e poi entrati nel codice dell'oggetto *Counter* condiviso, tutti leggono lo stesso stato ma non è ancora ben formato e si possono avere comportamenti non ben definiti. La parte in cui si accede ai dati condivisi, definito come *sezione critica*, è la parte in cui entrano più Thread in contemporanea e se vi entrano più Thread avremo degli errori; dobbiamo quindi impedirlo.

```
ExecutorService executor = Executors.newFixedThreadPool(1);

SimpleCounter counter = new UnsyncCounter();
List< Incrementer > incs = List.of(new Incrementer(counter),
  new Incrementer(counter), new Incrementer(counter),
  new Incrementer(counter));

executor.invokeAll(incs);

System.out.println("All done. Final state: " +
  counter.getState() + " (" + (end - time) + ")");
```

Applichiamo quindi la keyword *synchronized*, una parola chiave che applicata ad un blocco di istruzioni impedisce che sia percorso contemporaneamente da più di un Thread (non occorre importare classi o librerie per usarla). La keyword può decorare blocchi di istruzioni semplici o metodi.

Con questa implementazione che segue, il comportamento è ora corretto, essendo protetto l'accesso alla sezione critica:

```
class SyncCounter implements SimpleCounter {
  private int state = 0;
  syncronized public void add() {
    int current = state;
    try {
      TimeUnit.MILLISECONDS.sleep(
        Math.round(Math.random() * 100));
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    state = current + 1;
  }
}
```

Tutti i blocchi sincronizzati condividono lo stesso *monitor lock/intrinsic lock* associato all'oggetto stesso. Ogni oggetto ha un monitor, tranne primitivi e record; questi ultimi, quindi, non possono supportare la sincronizzazione.

Quando un Thread rilascia un monitor uscendo da un blocco synchronized, si ha una relazione di *happens-before* fra l'azione di rilascio del lock e ogni successiva acquisizione dello stesso.
Una forma alternativa di *synchronized* esplicita l'oggetto di cui effettuare la sincronizzazione.
Alcuni piccoli esempi di sintassi:

- *synchronized(that)* sincronizza esplicitamente il monitor dell'oggetto ritornato dall'espressione that
- *synchronized{}* equivalente a *synchronized(this) {}*

Tutti i monitor sono *reentrant*, quindi un thread può acquisire lo stesso *monitor lock* più volte senza entrare in *deadlock* con sé stesso.

Una relazione di happens-before è una garanzia forte fornita dal compilatore riguardo l'ordinamento dell'esecuzione delle istruzioni espresse dal codice. Fornire garanzie di questo tipo ostacola queste ottimizzazioni dato che, a causa della grande differenza di prestazioni fra il canale della memoria e la CPU, per motivi di efficienza il compilatore riordina aggressivamente l'ordine di esecuzione e richiede supporto dell'hardware, limitando la capacità del compilatore di ottimizzare il codice, riordinandone le istruzioni.
synchronized non è la soluzione a tutti i mali. In questo esempio, intendiamo modellare un attore che riceve un saluto e ricambia.

Per evitare che due attori si salutino contemporaneamente (sbattendo la testa), rendiamo synchronized i metodi di saluto, così un attore può essere salutato (metodo bow()) da un solo thread alla volta.

```
class SimpleFriend {
  private final String name;

  public synchronized void bow(SimpleFriend bower) {
    System.out.format("%s: %s" + " has bowed to me!%n",
      this.name, bower.getName());
    bower.bowBack(this);
  }

  public synchronized void bowBack(SimpleFriend bower) {
    System.out.format("%s: %s" + " has bowed back to me!%n",
      this.name, bower.getName());
  }
}
```

Tuttavia, il risultato non è quello che ci aspettiamo. In realtà, due attori che si salutano si bloccano l'uno con l'altro in un deadlock assolutamente classico (perché nessuno dei thread ottiene la risorsa dell'altro). Quello che succede è che Alphonse ottiene il lock su Gaston chiamando il suo *gaston.bow()*. Ma quando cerca di chiamare il metodo *bowBack()* su sé stesso, non può farlo perché Gaston a sua volta ha ottenuto il lock su di lui chiamando *alphonse.bow()*. Gaston è nella stessa situazione, e quindi i tue thread sono in deadlock.

```
public class SimpleFriends {

  public static void main(String[] args) {
    final SimpleFriend alphonse = new SimpleFriend("Alphonse");
    final SimpleFriend gaston = new SimpleFriend("Gaston");
    new Thread(() -> alphonse.bow(gaston)).start();
    new Thread(() -> gaston.bow(alphonse)).start();
  }
}
```

```
/**
* Causes the current thread to wait until another
* thread invokes the notify() method or the notifyAll()
* method for this object.
*/
void wait() throws InterruptedException;
```

Un'alternativa a synchronized è la *gestione esplicita del monitor di un oggetto.*
Possiamo introdurre l'eccezione nel metodo che fa attendere un thread esplicitamente:

Per operare sul monitor dell'oggetto il Thread deve "averlo a disposizione", asserendo la "proprietà" dell'oggetto. Un Thread può farlo:

- eseguendo un metodo *synchronized*
- eseguendo un blocco *synchronized* all'interno di un oggetto
- se l'oggetto è una *Class*, eseguendone un metodo *synchronized static*

Per svegliare un thread, pur non sapendo quale effettivamente chiamerà, usiamo il metodo void *notjfy();* similmente possiamo lanciarli tutti attraverso *notifyAll()*.

Prendiamo questa come classe di partenza *Named*:

```
class Named {
  public final String name;
  private boolean red = false;

  Named(String name) {
    this.name = name;
  }
```

```
synchronized void perform() throws InterruptedException {
  if (!red) {
    red = true;
    this.wait();
  } else {
    red = false;
    this.notify();
  }
}
```

Un solo thread alla volta può entrare in questo metodo. Se il flag è falso, viene posto a vero ed il thread si mette in attesa su questo oggetto (liberando il monitor). Se il flag è vero, viene posto a falso e un thread in attesa viene notificato che può proseguire.

Ora abbiamo un Runnable *Waiter* che usa due risorse della precedente classe Main dichiarate *final*:

```
class Waiter implements Runnable {
  private final Named first, second;

  Waiter(Named first, Named second) {
    this.first = first;
    this.second = second;
  }
```

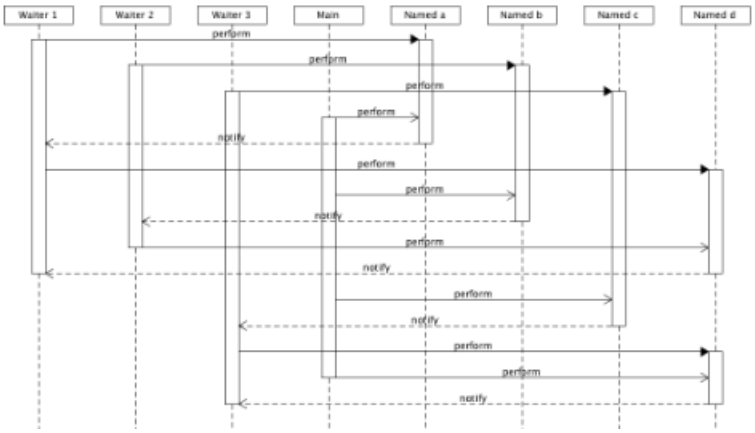
```
var thread = Thread.currentThread().getName();
System.out.println(thread + " waiting on " + first.name);
String doing = first.name;
try {
  first.perform();
  System.out.println(thread + " signalled on " + first.name);
  System.out.println(thread + " waiting on " + second.name);
  doing = second.name;
  second.perform();
} catch (InterruptedException e) {
  System.out.println(thread + " interrupted on " + doing);
}
System.out.println(thread + " signalled on " + second.name);
```

Il suo obiettivo è eseguire *perform()* prima su una risorsa, poi sulla seconda:

Il main crea quattro risorse e tre threads, che condividono la quarta risorsa. Esegue perform() sulle risorse una dopo l'altra, andando a liberare i thread che erano in attesa di sincronizzazione. Infine, richiama performAll() sull'ultima risorsa, liberando i thread che erano là bloccati.

```
Named a = new Named("a"), b = new Named("b"),
c = new Named("c"), d = new Named("d");
new Thread(new Waiter(a, d), "Waiter 1").start();
new Thread(new Waiter(b, d), "Waiter 2").start();
new Thread(new Waiter(c, d), "Waiter 3").start();
try {
  a.perform();
  b.perform();
  c.perform();
  d.performAll();
} catch (InterruptedException e) { e.printStackTrace(); }
```

L'ordine delle operazioni, quando non mediato dai synchronized, è assolutamente casuale. I Waiter si appropriano della prima risorsa, ma devono aspettare Main che li fa avanzare. A questo punto, uno di loro ottiene la seconda risorsa mentre l'altro aspetta. Ciò può accadere in maniera indefinita, quindi rappresenta un rischio.



Il non determinismo determina ogni volta degli scambi diversi, usando tutto il numero corretto di volte.
Un thread può essere svegliato da un wait senza nessun notify; ciò è detta "*spurious wakeup*" (la documentazione stessa incoraggia tale comportamento).
synchronized crea un blocco implicito, mentre *wait()* costringe a gestire lo stato del blocco; tuttavia, alcune volte, abbiamo bisogno di controllare esplicitamente le condizioni di blocco/sblocco.

L'uso di un Lock ci permette di slegare l'acquisizione ed il rilascio di una risorsa dalla struttura lessicale in cui questo avviene. Questo perché non è legato all'esecuzione di un blocco di codice come sono invece synchronized e wait().

```
/**
* Lock implementations provide more extensive locking
* operations than can be obtained using synchronized
* methods and statements.
*
*/
public interface Lock;
```

```
class LockedFriend {
  private final String name;
  private final Lock lock = new ReentrantLock();

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

  public String getName() {
    return this.name;
  }
```

L'acquisizione avviene poi con il metodo void *lock(),* che è chiamata bloccante se il lock non è disponibile.
La chiamata del metodo void *unlock()* ha l'effetto di sbloccare un thread tra quelli che attendevano di acquisire un lock. Il metodo booleano *tryLock()* acquisisce il lock solo se è libero quando è stato invocato.

Scritto da Gabriel

Andiamo ora a risolvere il problema dei due attori che si salutano: proviamo ad acquisire entrambi i lock; siamo ora in grado di rilasciarli in caso di acquisizione parziale.

```
public boolean impendingBow(LockedFriend bower) {
  boolean myLock = false, yourLock = false;
  try {
    myLock = lock.tryLock();
    yourLock = bower.lock.tryLock();
  } finally {
    if (!(myLock && yourLock)) {
      if (myLock) { lock.unlock();}
      if (yourLock) { bower.lock.unlock(); }
    }
  }
  return myLock && yourLock;
}
```

```
public void bow(LockedFriend bower) {
  if (impendingBow(bower)) {
    try {
      out.format("%s: %s has" + " bowed to me!%n",
        this.name, bower.getName());
      bower.bowBack(this);
    } finally { lock.unlock(); bower.lock.unlock(); }
  } else {
    out.format("%s: %s started to bow to me, but saw that"
      + " I was already bowing to him.\n",
      this.name, bower.getName());
  }
}
```

Se impendingBow() ritorna *true*, abbiamo entrambi i lock (e quindi dobbiamo rilasciarli). Non siamo legati ai monitor impliciti o a lavorare all'interno di un solo blocco di codice. Se impendingBow() ritorna *false*, abbiamo evitato un deadlock: avendo rilasciato l'acquisizione parziale, abbiamo probabilmente permesso all'altro thread di completare la sua.

L'esempio con relativo *main()* è preso da:
https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html

Introduciamo il pattern architetturale dei *Producer/Consumer* (che si avvicina al concetto di coroutines):

- Producers, threads che producono dati da elaborare
- Consumers, threads che elaborano i dati prodotti

Con un Lock controlliamo manualmente una sezione critica, in cui il Consumatore ammette un solo Thread alla volta nella sezione critica. Il Lock, implicitamente, gestisce la coda dei Produttori in attesa.

Quella del lock può essere (se richiesto) una coda di attesa. Quello del monitor di un oggetto invece è sempre un insieme di thread in attesa. Nota: un *implicit lock* permette di acquisire automaticamente le risorse necessarie in quel contesto di esecuzione qualora si possieda il lock di riferimento, mentre un *re-entrant lock* si ha quando un processo prende possesso del lock più volte senza bloccarsi (es. utile: l'accesso di dati iterativamente in un grafo; vogliamo fare in modo di passare sullo stesso vertice e continuare a computare senza problemi).

Quindi introduciamo un *reentrantLock*, con caratteristiche equivalenti ad un *implicit lock*, ma può essere controllato manualmente. Se viene costruito come *fair* il Thread che riceve il lock è sempre quello che ha aspettato di più; si riduce quindi il rischio di *starvation* a costo di una prestazione inferiore (perché si deve mantenere una lista ordinata). Implementiamo quindi una coda di stampa in cui, rispetto al *BlockingQueue*, gestiamo direttamente l'accesso alla sezione critica.

```
public class PrintQueue implements Printer {
  private final Lock queueLock = new ReentrantLock();

  public void printJob(Object document) {
    queueLock.lock();
    try {
      Long duration = (long) (Math.random() * 10000);
      Thread.sleep(duration);
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally { queueLock.unlock(); }
  }
}
```

```
public static void main(String args[]) {
  PrintQueue printQueue = new PrintQueue();
  Thread thread[] = new Thread[10];
  for (int i = 0; i < 10; i++) {
    thread[i] = new Thread(new Job(printQueue),
      "Thread " + i); }
  for (int i=0; i < 10; i++) {
    thread[i].start(); }
  }
```

Non tutti i Thread che cercano di acquisire il lock sono uguali, dato che possono avere semantiche diverse e condizioni differenti. Una *Condition* separa l'accodamento in attesa dal processo di lock che controlla l'attesa stessa, gestendo quindi su un solo lock più condizioni di attesa distinte. Esso rappresenta la gestione dell'accesso alla stessa sezione critica in condizioni di blocco/sblocco differenti.

Un oggetto Condition viene fornita dal lock su cui deve sussistere e ciascuna consente di gestire un insieme distinto di Thread in attesa. Vengono creati con *newCondition()*, e abbiamo una serie di metodi utili come *await()* (che fa attendere il thread corrente), *signal()/signalAll()* (che sveglia uno o più thread).

```
/**
 * Bounded, non thread-safe random source of characters
 */
class CharSource {
 public boolean hasMoreLines()
 public Optional< String > getLine()
```

Abbiamo poi una sorgente di linee di testo:

```
class Buffer {
private LinkedList< String > buffer;
  private int maxSize;
  private ReentrantLock lock;
  private Condition lines, space;
  private boolean pendingLines;

  public Buffer(int maxSize) {
    this.maxSize = maxSize; pendingLines = true;
    buffer = new LinkedList<>();
    lock = new ReentrantLock();
    lines = lock.newCondition();
    space = lock.newCondition();
  }
```

Questo a fianco è un Buffer. Crea un lock, per bloccare l'accesso alle zone critiche, e ne ottiene due Condition: una sarà usata per attendere la presenza di nuove linee, l'altra per mettere in attesa i thread che non trovano dati da consumare.

```
public void insert(String line) {
  lock.lock();
  try {
    while (buffer.size() == maxSize) {
      space.await();
    }
    buffer.offer(line);
    out.printf("%s: Inserted Line: %d\n",
    Thread.currentThread().getName(), buffer.size());
    lines.signalAll();
  } catch (InterruptedException e) {
    e.printStackTrace();
  } finally { lock.unlock(); }
}
```

Per inserire una linea, prima di tutto acquisiamo il lock per la sezione critica; se non c'è spazio, ci mettiamo in attesa anche sul lock dello spazio. Infine, aggiungiamo la riga al buffer e segnaliamo chi attendeva linee che ce ne sono di disponibili.

Per ottenere una riga, prima di tutto acquisiamo il lock per la sezione critica; se non ci sono linee, attendiamo sulla condizione che ci siano righe a disposizione. Se ce ne sono, ne otteniamo una; potrebbe però essere vuota, perché un altro thread l'ha presa prima di noi. Infine, segnaliamo disponibilità di spazio e rilasciamo il lock critico.

```
public Optional< String > get() {
  Optional< String > line = Optional.empty();
  lock.lock();
  try {
    while ((buffer.size() == 0) && (hasPendingLines())) {
      lines.await();
    }
    if (hasPendingLines()) {
      line = Optional.ofNullable(buffer.poll());
    space.signalAll(); }
  } catch (InterruptedException e) { e.printStackTrace(); }
  } finally { lock.unlock(); }
  return line;
}
```

```
public boolean hasPendingLines() {
  return pendingLines || buffer.size() > 0;
}

public void setPendingLines(boolean pendingLines) {
  this.pendingLines = pendingLines;
}
```

Questi due metodi ci permettono di controllare se ci sono linee pendenti, o di impostare la loro disponibilità.

Il Producer è un Runnable che prende una riga dalla sorgente, e la mette nel buffer, segnalando la presenza di nuove righe. E la loro assenza quando la sorgente è consumata. Il flag *pendingLines* del Buffer permette al produttore di segnalare la propria presenza ai consumatori, in modo che siano rassicurati dell'arrivo di nuove linee.

```
class Producer implements Runnable {
  private CharSource source;
  private Buffer buffer;

  @Override
  public void run() {
    buffer.setPendingLines(true);
    while (source.hasMoreLines())
      source.getLine().ifPresent((line) -> {
        buffer.insert(line); randomWait(50);
      });
    buffer.setPendingLines(false);
  }
}
```

```
class Consumer implements Runnable {
  private Buffer buffer;

  @Override
  public void run() {
    while (buffer.hasPendingLines())
    buffer.get().ifPresent((line) -> process(line));
  }

  private void process(String line) {
    LockedBuffer.randomWait(250);
  }
}
```

Il Consumer è un altro Runnable che prende una riga dal buffer e ci spende sopra un breve lasso di tempo. Quando non trova più la segnalazione di nuove righe (che include sia linee presenti nel buffer, sia la presenza di un produttore che ne aggiunga) chiude l'esecuzione.

Infine, il main fa partire tutti i thread. Siccome non usiamo Executor, l'algoritmo viene eseguito e la JVM termina quando tutti i thread terminano.

```
public static void main(String[] args) {
  CharSource source = new CharSource(100, 100);
  Buffer buffer = new Buffer(20);
  Thread producer = new Thread(new Producer(source, buffer),
    "producer");
  Thread[] consumers = new Thread[] {
    new Thread(new Consumer(buffer)),
    new Thread(new Consumer(buffer)),
    new Thread(new Consumer(buffer)) };
  producer.start();
  for (Thread t : consumers) t.start();
}
```

Maneggiando direttamente i Lock si chiede al sistema di delegarci un notevole potere, ed insieme ne riceviamo una corrispondente responsabilità.
Per controllare l'accesso ad un insieme omogeneo di risorse usiamo i *semafori*, simili ai lock tengono dei conteggi al posto degli stati libero/occupato.

```
/**
* Creates a Semaphore with the given number of permits and
* the given fairness setting.
*
*/
public Semaphore(int permits, boolean fair)
```

Se inizializzato come *fair*, l'ordinamento dei Thread in attesa è garantito come FIFO. Il costo è giustificato quando il semaforo regola l'accesso ad un insieme di risorse; in caso di uso diverso, un semaforo non fair è molto più efficiente. Ad ogni acquisizione, diminuisce il numero di permessi e, ad ogni rilascio, il numero viene aumentato. Ciò si realizza con *void acquire()* oppure *void acquire(int permits).* Con la stessa sintassi similmente si ha *release() e release(int permits)*.

Il valore iniziale del semaforo non è un limite, può essere superato e può anche essere negativo quando inizializzato. A differenza di un lock, può essere rilasciato da un Thread diverso da quello acquisito.
È possibile ridurre il numero di permessi disponibili, togliendone alcuni (con il metodo *reducePermits(int reduction)*):

Abbiamo quindi due possibili eccezioni lanciate tra le varie di Semaphore:

- lanciare *InterruptedException* se il thread viene interrotto durante l'attesa
- lanciare *IllegalArgumentException* se il parametro è negativo

Si ha poi un metodo di acquisizione del permesso del semaforo, se ce ne sta uno disponibile (*tryAcquire()*), che può anche non avere argomenti o averne come riportato qui.
tryAcquire ritorna immediatamente *false* se non ha ottenuto un permesso. *È quindi in grado di violare la fairness del semaforo.*

```
/**
* Acquires the given number of permits from this semaphore,
* if all become available within the given waiting time and
* the current thread has not been interrupted.
*
*/
public boolean tryAcquire(int permits, long timeout,
                          TimeUnit unit)
```

```
class MultiPrintQueue implements Printer {
  private Semaphore semaphore;
  private boolean[] freePrinters;
  private ReentrantLock lockPrinters;

  public MultiPrintQueue() {
    semaphore = new Semaphore(3);
    freePrinters = new boolean[] { true, true, true };
    lockPrinters = new ReentrantLock();
  }
```

Questa implementazione di Printer si basa su un semaforo per contare le stampanti libere ed un array di boolean per mantenere lo stato delle singole stampanti.

L'esecuzione di una stampa richiede di acquisire il semaforo (che attende una stampante libera se non ce n'è), ottenere la stampante da assegnare, effettuare il lavoro, liberare la stampante e quindi il semaforo.

```
public void printJob(Object document) {
  try {
    semaphore.acquire();
    int assignedPrinter = getPrinter();
    Long duration = (long) (Math.random() * 10000);
    TimeUnit.MILLISECONDS.sleep(duration);
    freePrinters[assignedPrinter] = true;
  } catch (InterruptedException e) { e.printStackTrace(); }
  } finally { semaphore.release(); }
}
```

```
int getPrinter() {
  int res = -1;
  try {
    lockPrinters.lock();
    for (int i = 0; i < freePrinters.length; i++) {
      if (freePrinters[i]) {
        res=i; freePrinters[i]=false;
        break;
      }
    }
  } catch (Exception e) { e.printStackTrace();
  } finally { lockPrinters.unlock(); }
  return res;
}
```

La selezione di una stampante libera richiede, all'interno di una sezione critica, di cercare un valore *true* nell'array dello stato delle stampanti. Abbiamo la garanzia che ce ne sia almeno uno perché siamo protetti dal semaforo. Trovata la stampante libera, la segniamo occupata e ritorniamo il suo indice al chiamante uscendo dalla sezione critica.

Lezione 11 – Dati thread safe

Nell'utilizzo di dati concorrenti, in particolar modo tra Thread diversi, attenzione agli oggetti in stato mutabile e condiviso. Per esempio in:

```
public class Adder {

int target = 0;

public void add() {
    ...
    t1.start();
    t2.start();
    t3.start();
  }
}
```

```
var t1 = new Thread(() -> {
  for (int i = 0; i < 100000; i++) { target +=1; }
});
var t2=...
var t3=...
```

Allo scopo di questo test, è più semplice ripetere la definizione per ciascun thread perché l'obiettivo è riferire alla stessa variabile condivisa. Le tre lambda scritte in questo modo sono closures (cioè, si chiudono) sulla variabile target lessicalmente identica.

Testando su:

```
@Test void test() throws InterruptedException {
  var adder=new Adder();
  adder.add();
  Thread.sleep(1000);
  assertEquals(300000, adder.target);
}
```

La concorrenza ci porta al non determinismo. Il problema di condividere l'accesso a dati non è quindi solo quello del deadlock, ma anche quello della correttezza del risultato. Va notato come già i passaggi di compilazione, interpretazione e JITting introducono una indeterminatezza sul reale ordine di esecuzione delle istruzioni, anche nel caso del singolo thread. Inoltre, lo stesso hardware spesso riordina l'esecuzione delle istruzioni per ottimizzare l'uso delle risorse. Quindi, in linea generale non ci si può basare sul codice sorgente per interpretare alcuni dettagli del comportamento di un programma, tantomeno in ambiente concorrente.

Una struttura dati non thread-safe non consente a più thread di operare contemporaneamente.

- nel migliore dei casi lancia una *java.util.ConcurrentModificationException*
- nel caso intermedio lo stato diventa inconsistente
- nel peggiore dei casi ottengo un'altra eccezione (per es. in determinati (rari) casi HashMap#put è in grado di lanciare un IndexOutOfBound.
-

Per esempio, vediamo l'attraversamento della lista attraverso un *Runnable* e poi averne uno, che aggiunge elementi alla lista con il metodo *add()*:

```
list.iterator().forEachRemaining(el -> {
  out.println(el);
  try {
    Thread.sleep(250);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
});
try {
  Thread.sleep(300);
} catch (InterruptedException e) {
  e.printStackTrace();
}
list.add("d");
```

```
List< String > list = new ArrayList< String >();
list.add("a");
list.add("b");
list.add("c");
var t1 = new Thread(new ListTraverser(list));
var t2 = new Thread(new ListUpdater(list));
t1.start();
t2.start();
Thread.sleep(1000);
assertEquals(4, list.size());
```

Segue il test del funzionamento della lista, avviando alcuni thread e poi verificando i thread avviati (in particolare ArrayList dichiara (nella documentazione) che il suo iteratore lancia una *java.util.concurrentModificationException* se la collezione viene modificata durante l'attraversamento):

Di fatto quindi per condividere dati fra più thread, abbiamo bisogno di strutture dati thread-safe. Se quindi il caso d'uso riguarda l'accesso ad una variabile scalare, come ad esempio l'accesso ad un contatore, usiamo il package *java.concurrent.atomic*, con alcune classi di riferimento:

tipo	singolo
Integer	`AtomicInteger`
Long	`AtomicLong`
Object	`AtomicReference`

tipo	array
Integer	`AtomicIntegerArray`
Long	`AtomicLongArray`
Object	`AtomicReferenceArray`

Le classi di questo package sono particolarmente efficienti, poiché in caso di modifica concorrente del dato che rappresentano, usano le funzionalità fornite dall'hardware.

Esse sono garantite come thread-safe, sapendo che le operazioni sono eseguite in modo atomico e la modifica (quasi sempre) non blocca il thread eseguito; questo avviene se c'è il supporto dell'hardware, come ad esempio le istruzioni *CAS (Compare and swap)*, operazioni unitarie e che permettono l'esecuzione concorrente, altrimenti dovremmo usare metodi più convenzionali, che bloccano i thread e quindi meno efficienti (per esempio il controllo sul lock, deve essere *synchronized* e poi successivamente si eseguono operazioni dopo aver eseguito un certo controllo.

Il discorso è delicato; infatti il thread non è mai del tutto bloccato nell'esecuzione del metodo, cerca sempre di eseguire il codice CAS finché non riesce). Un esempio di operazioni CAS sono quelle della classe *Atomic*. Buona idea a: https://jenkov.com/tutorials/java-concurrency/compare-and-swap.html

Ad esempio, l'implementazione di thread con garanzia di esecuzione atomica; ora l'istruzione appare anche nel sorgente come unitaria: incrementa e ritorna il nuovo valore. La documentazione spiega le garanzie di atomicità che sono fornite.

```
public class AtomicAdder {

  public AtomicInteger target = new AtomicInteger(0);

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

```
var t1 = new Thread(() -> {
for (int i = 0; i < 100000; i++) {
  target.incrementAndGet(); }
});
```

La terza chiamata di incrementAndGet potenzialmente potrebbe rimanere bloccata, ma nella pratica la risposta è talmente veloce che è estremamente difficile (praticamente impossibile, se l'esecuzione viene implementata con un solo opcode) che le due incrementeAndGet si accavallino. Qualora anche avvenisse, a livello hardware il costo di sincronizzazione sarebbe molto basso.

Atomicamente abbiamo dei confronti con *compareAndset*, che verifica che a seguito della computazione, si sia ottenuto il valore atteso, a seguito di aggiornamento:

```
/**
* Atomically sets the value to the given updated
* value if the current value == the expected value.
*
* @param expect the expected value
* @param update the new value
* @return true if successful. False return indicates
* that the actual value was not equal to the expected value.
*/
public final boolean compareAndSet(long expect, long update)
```

Marcando poi l'oggetto come utilizzato, con un bit di mark aggiornato atomicamente, sulla base del tipo di riferimento (similmente, si può associare un timestamp ed avere lo stesso funzionamento e caratteristiche con *AtomicStampedReference<V>*):

```
/**
* An AtomicMarkableReference maintains an object
* reference along with a mark bit, that can be
* updated atomically.
*
* Implementation note : This implementation maintains
* markable references by creating internal objects
* representing "boxed" [reference, boolean] pairs.
*/
public class AtomicMarkableReference< V >;
```

Per indicare che una variabile deve essere sempre letta dalla memoria principale è *volatile*, non dalle cache intermedie. Quindi normalmente nelle architetture hardware abbiamo più CPU che colloquiano tra loro velocizzando codice ed esecuzione. Tuttavia, proprio a causa della contemporaneità di operazioni su una stessa esecuzione, i thread possono vedere valori diversi (problema della "visibilità del codice scritto") e si hanno ottimizzazioni e compromessi necessari in architetture multiprocessore/multithread.
Esempio in cui si presenta: interazione con codice nativo interfacciato via JNI che riporta nulla memoria principale una misura che cambia di continuo (es. antenne RFID).

Una variabile dichiarata "*volatile*" viene sempre letta dalla memoria principale in modo da garantire la visibilità dell'ultima scrittura. Secondo la specifica della JVM, *volatile* stabilisce una relazione di *happens-before* in determinati casi di accesso alla variabile.

In particolare, la documentazione di Java dice che "A write to a volatile field happens-before every subsequent read of that field." La garanzia che fornisce *volatile* è il fatto che il programma è corretto solo se *nessun thread scrive nella variabile volatile un valore dipendente dal valore che è appena stato letto dalla stessa variabile*. *atomic* generalizza e semplifica alcuni usi di *volatile*. Da usare con cautela.

Infatti, in alcuni casi lo stato del calcolo può generare inconsistenze sull'ordine delle operazioni, come nel codice qui sotto dove attendiamo che il servizio abbia esaurito ogni task accodato per ogni stream, controllando ciascuno di questi per mezzo dell'esecutore e controllando lo stato finale del contatore.

```
class VolatileHolder {
  volatile int counter = 0;
}
```

```
@Test
public void volatileCounter() {
  VolatileHolder holder = new VolatileHolder();

  ExecutorService executor = Executors.newFixedThreadPool(4);
  IntStream.range(0, 10000).forEach(i ->
executor.submit(() -> holder.counter++));
  awaitDone(executor);

  assertEquals(10000, holder.counter);
}
```

Nel package *java.util.concurrent* vi sono quindi versioni ottimizzate delle collezioni più comuni, con implementazioni specifiche, ad esempio: *Collections.syncronizedMap(newHashMap()).*
Rispetto alla sincronizzazione della completa struttura dati, queste classi limitano la sincronizzazione alle sole sezioni critiche, in modo da ottenere complessivamente una maggiore efficienza.
L'interfaccia *ConcurrentMap* estende *Map* avendo garanzie di atomicità ed ordinamento delle operazioni.

```
/**
* A Map providing thread safety and atomicity
* guarantees.
*/
public interface ConcurrentMap< K,V >
  extends Map< K,V >
```

Specificando che nessun altro thread può mettere il suo valore se non ben associato:

```
/**
* If the specified key is not already associated with a
* value, associate it with the given value.
* The action is performed atomically.
*/
V putIfAbsent(K key, V Value)
```

```
/**
* Replaces the entry for a key only if currently mapped
* to some value.
* The action is performed atomically.
*/
V replace(K key, V Value)
```

Rimpiazzando magari un valore se il valore esistente è quello che ci aspettiamo (con due varianti, qui la prima e sotto la seconda):

```
/**
* Replaces the entry for a key only if currently mapped
* to a given value.
* The action is performed atomically.
*/
V replace(K key, V oldValue, V newValue)
```

Analogamente abbiamo *ConcurrentNavigableMap* con *NavigableMap*, oppure *ConcurrentHashMap*, offrendo metodi come *reduce, search, foreach*, che possono operare su tutte le chiavi, suddividendo il lavoro su più thread (multithreading).

L'operazione di *reduce* divide la mappa in parti, gestendo i risultati su ciascun thread diverso. Essa riduce la mappa ad un risultato unico. Vanno fornite due funzioni: una per trasformare le coppie chiave/valore nel tipo del risultato, e l'altra per sommare i risultati parziali. Il tipo del risultato deve quindi essere dotato di un'operazione di somma con le consuete proprietà commutativa e associativa.

```
/**
* Returns the result of accumulating the given
* transformation of all (key, value) pairs using the
* given reducer to combine values, or null if none.
*
* @param the elements needed to switch to parallel
* @param the transformation for an element
* @param a commutative associative combining function
*/
public < U > U reduce(long parallelismThreshold,
  BiFunction< ? super K,? super V,? extends U> transformer,
  BiFunction< ? super U,? super U,? extends U> reducer)
```

```
Random rnd = new Random();
ConcurrentHashMap< String, Long > map =
  new ConcurrentHashMap< String, Long>();
IntStream.range(0, 10000)
  .forEach(i -> map.put("k" + i,
    new Long(rnd.nextInt(1000))));
```

Per provare il funzionamento di queste istruzioni, costruiamo una mappa di *long* casuali.

Avviamo la sua riduzione, suggerendo tramite l'apposito parametro di usare il parallelismo. La somma parallela qui ci mette 8 secondi, quella seriale ben 55.

```
long start = System.currentTimeMillis();
Long parres = map.reduceEntries(500,
  entry -> entry.getValue(), (a, b) -> a + b);
long partime = System.currentTimeMillis() - start;
```

```
/**
 * Returns a non-null result from applying the given
 * search function on each (key, value), or null if none.
 * Upon success, further element processing is suppressed.
 *
 * @param the elements needed to switch to parallel
 * @param a search function, that returns non-null on
 *success
 */
public < U > U search(long parallelismThreshold,
  BiFunction< ? super K,? super V,? extends U> searchFunction)
```

La funzione *search* permette di applicare una ricerca parallela nella mappa. Il risultato è il primo non nullo ritornato dalla funzione di chiave e valore. La ricerca è parallela, ma trovato il risultato tutti i thread di ricerca vengono fermati.

Infine, la funzione forEach permette di eseguire un effetto collaterale per ciascuna coppia chiave/valore.

```
/**
 * Performs the given action for each (key, value).
 *
 * @param the elements needed to switch to parallel
 * @param the action (can have side-effects)
 */
public void forEach(long parallelismThreshold,
  BiConsumer< ? super K,? super V> action)
```

Le funzioni usate nei metodi di trasformazione non devono dipendere oltre che dall'ordinamento anche dallo stato condiviso nel calcolo (senza *side-effects*), dando varie versioni di un algoritmo, con una trasformazione opzionale prima dell'uso del valore, iterazione su chiavi/valori o con risultati primitivi.
Le operazioni di riduzione, ricerca ed esecuzione di effetti non sono atomiche nel loro complesso, ma ogni coppia chiave-valore non nulla ha una garanzia di *happens-before* con il suo uso nell'iterazione.

Segue l'interfaccia *BlockingQueue*, che aggiunge alla classica *Queue* metodi di scelta nelle operazioni di accodamento/prelievo, richiedendo un certo comportamento in caso di esecuzione concorrente.
Vediamo i vari metodi (per risultato negativo si intende elemento non disponibile per l'operazione):

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

Lettura

metodo	risultato negativo
`element()`	eccezione
`peek()`	`null`

Attenzione che il risultato non è definito (non interviene un'eccezione) se la coda viene modificata durante l'operazione: gli elementi accodati potrebbero essere riportati nella collezione, oppure ignorati.

Questa operazione è più efficiente che un ciclo di poll fino ad esaurimento, ma non è atomica.

```
/**
 * Removes all available elements from this
 * queue and adds them to the given collection.
 * This operation may be more efficient than
 * repeatedly polling this queue.
 *
 * @param c the collection to transfer elements into
 * @return the number of elements transferred
 */
int drainTo(Collection< ? super E > c)
```

Conclusione lezione 11 ed inizio lezione 12 – Parallel Streams

BlockingQueue implementa naturalmente sistemi Produttore-Consumatore. La sua definizione è:
"Una coda che supporta le operazioni che bloccano in attesa che la coda si riempia durante il recupero di un elemento e si bloccano in attesa che si abbia altro spazio disponibile durante l'archiviazione di un elemento."

Implementiamo per esempio un sistema costruito in questo modo: la Main class carica comandi nella classe Printer; la classe Printer mantiene una coda dei lavori da effettuare, ed istanzia un numero di Driver che pescano dalla coda ed eseguono i lavori. La classe Main crea la stampante e una decina di thread che accodano un job sulla stampante stessa.

```
Printer concurrent = new ConcurrentPrinter(8);
Thread thread[] = new Thread[10];
out.println("Preparing...");
IntStream.range(0, 10).forEach((i) -> thread[i] =
  new Thread(() -> {
    out.println("Queueing job " + i);
    concurrent.printJob(new Object());
    out.println("Job " + i + " queued");
  }));
out.println("Starting.");
for (int i = 0; i < 10; i++) { thread[i].start(); }
```

```
ConcurrentPrinter(int printers) {
  // limit printers to effective cores
  size = printers < cores ? printers : cores;
  // size and build the queue
  queue = new LinkedBlockingQueue< PrintJob >(QUEUE_SIZE);
  // start the executor
  executor = Executors.newFixedThreadPool(size);
  // start drivers
  IntStream.range(0, size).forEach((a) ->
    executor.execute(new PrinterDriver(queue)));
}
```

La stampante inizializza nel costruttore la coda, l'Executor e avvia i driver.

Un PrintJob è una semplice value class che tiene il tempo di quando è stata creata per poter misurare il tempo di attesa in coda.

```
@Override
public void printJob(Object document) {
  try {
    queue.put(new PrintJob(document));
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

```
public void run() {
  try {
    while (true) {
      PrintJob job = queue.take();
      out.printf(...);
      int duration = rnd.nextInt(2500);
      Thread.sleep(duration);
      out.printf(...);
    }
  } catch (InterruptedException ex) {
    out.println("Printer shutting down.");
  }
}
```

Il driver si mette in attesa di un elemento dalla coda, e simula la sua esecuzione in stampa. I messaggi danno conto del tempo d'attesa del job in coda.

Si consiglia di implementare una serie di cose (presenti nelle slide):
Esercizio 1: implementare un SerialPrinter che crea un solo driver ed usa un solo thread
(fatto, incluso sotto forma di file su Mega).

Esercizio 2: Casualmente, il numero di job che vengono accodati è pari a ConcurrentPrinter.QUEUE_SIZE.
È facile immaginare cosa succede se sono di meno. Cosa avviene se sono di più?

- Se sono di più, i thread rimarranno in attesa attiva finché non si libera il buffer e qualcuno di questi può intervenire occupandosi e/o prendendo risorse.

Esercizio 3: La classe PrinterOperator accoda i job tramite threads separati.
Cosa succede, nelle condizioni dell'esercizio 2, se invece la classe accodasse direttamente i job?

- Avendo più thread ed accodando direttamente i job, si avrebbe un giusto ordine di esecuzione/blocco dei singoli thread, evitando quindi l'esecuzione infinita.
  A mio dire si usa l'interfaccia qui appena vista, BlockingQueue, tale da eseguire correttamente queste operazioni.

Esercizio 4: La classe PrinterOperator così come è scritta non termina: la JVM non si chiude perché l'Executor del ConcurrentPrinter non viene chiuso. Come si può fare per permettergli di chiudere correttamente, dopo aver eseguito tutti i job?
Nota: questo è un esercizio di design; non è detto che sia risolvibile nel design attuale.

- Per far chiudere correttamente questa classe, sarebbe per me necessario quantomeno l'uso di una struttura in grado di gestire insieme tutti questi thread, garantendo una fairness di esecuzione ed acquisizione del semaforo. Sarebbe infatti necessario garantire una giusta attesa ma soprattutto certezza di azione ed esecuzione. Si dovrebbe implementare una struttura che:
  permette di meglio sfruttare la struttura FixedThreadPool
  un'idea implementativa come BlockingQueue ma adatta a contesti come semafori o altro

Esistono altre varianti di Queue come:

`TransferQueue`: interfaccia per una coda in cui i produttori aspettano i consumatori
`BlockingDeque`: interfaccia che permette di prendere un elemento dalla coda o dalla testa

`PriorityBlockingQueue`: coda ordinata per priorità
`DelayQueue`: un elemento non può essere preso prima di un ritardo impostato
`SynchronousQueue`: ogni produttore deve attendere un consumatore (capacità nulla)

`ArrayBlockingQueue`: implementazione basata su array, con possibilità di *fairness*
`LinkedBlockingDeque`,
`LinkedBlockingQueue`,
`LinkedTransferQueue`: implementazioni basate su liste collegate

Abbiamo poi le *thread local variables*, in cui si tenta un altro approccio, cioè garantire che ogni Thread abbia un valore separato ed indipendente rispetto agli altri.
Generalmente sono istanziati come: *public class ThreadLocal <T>*
Una variabile *ThreadLocal* esiste in una copia differente ed indipendente per ciascun Thread che attraversa la sua dichiarazione. Qui sotto, il Supplier permette l'inizializzazione a partire da una strategia esterna.

```
/**
 * Creates a thread local variable. The initial value of
 * the variable is determined by invoking the get method
 * on the Supplier.
 *
 */
static < S > ThreadLocal< S >
  withInitial(Supplier< ? extends S > supplier)
```

Similmente a prima abbiamo metodi che ritornano una copia della variabile local-thread (*get()*), rimozione (*remove()*), impostazione (*set()*), ritorno del valore iniziale (*initialValue()*).
Qui a fianco, il valore *nextId* è globale; ogni thread accede sempre allo stesso. Il contatore counter invece è privato di ciascun thread.

```
class LocalVar {
  private static final var nextId = new AtomicInteger(0);

  ThreadLocal< Integer > counter;

  LocalVar() {
    counter = ThreadLocal.withInitial(() ->
      nextId.incrementAndGet());
  }

  Integer get() { return counter.get(); }
}
```

```
class LocalReader implements Runnable {
  private final LocalVar var;
  private final int item;

  @Override
  public void run() {
    out.println(Thread.currentThread().getName() +
      ", item " + item + ": read " + var.get());
  }
}
```

Questo Runnable legge e stampa il valore della variabile ThreadLocal:

Questo main lancia diversi Runnable che condividono la medesima istanza di LocalVar. Eppure, ciascuno di loro vi legge un valore diverso per effetto della variabile ThreadLocal.

```
ExecutorService executor = Executors.newFixedThreadPool(4);
LocalVar var = new LocalVar();
IntStream.range(0, 20).forEach((a) ->
  executor.execute(new LocalReader(var, a)));
executor.shutdown();
```

Queste variabili sono molto simili alle variabili globali, per cui quando i thread hanno finito di utilizzare i dati, dovrebbero rimuovere il loro oggetto thread local. Questo può condurre a memory leak, dunque.

Lo *Stream* diventa a tutti gli effetti un oggetto, trattabile come tale, dove la promozione va verso un approccio funzionale al trattamento iterativo dei dati su un insieme di oggetti, non sapendo a priori quanti oggetti possano essere (cardinalità potenzialmente infinita). In cambio di alcune restrizioni sulle operazioni possibili e della cessione del controllo sull'iterazione, otteniamo:

- una API per comporre passi di elaborazione riusabili su di una successione di oggetti
- una reificazione della trasformazione, che ne permette il riuso su sorgenti diverse ed il test in isolamento
- la promozione di un approccio funzionale al trattamento iterativo dei dati
- la disponibilità di uno stile espressivo e dichiarativo per descrivere una ampia classe di elaborazioni
- un modello di esecuzione in cui esecuzione sequenziale e parallela non richiedono modifiche al codice

Cosa cambia quindi tra uno Stream e le Collezioni?

- Una Collezione ha come caratteristica buona la performance di accesso ai contenuti e l'algoritmo che li usa è estraneo a queste considerazioni
- Uno Stream richiede la definizione dell'algoritmo di calcolo che avverrà sopra i suoi elementi, avendo un certo controllo, usato per prendere decisioni sull'esecuzione.

Nota: Gli stream non possono essere riutilizzati, potrebbero infatti richiamare *IllegalStateException.*
La sorgente di uno Stream può dichiarare alcune caratteristiche che gli operatori intermedi possono verificare e che l'operazione terminale usa per prendere decisioni sull'esecuzione (che non avviene mai prima dell'attivazione dell'operazione terminale).
Descriviamo quindi gli *stream flags:*

CONCURRENT	Parallelizzabile
DISTINCT	Elementi distinti
IMMUTABLE	Immutabile durante il consumo
SORTED	Ordinamento definito
SUBSIZED	Suddivisioni di dimensione nota

NONNULL	Elementi non nulli
ORDERED	Elementi ordinati
SIZED	Dimensione nota

L'operazione terminale ha piena visibilità di quali siano le caratteristiche della pipeline di esecuzione e può quindi prendere decisioni in merito. Nell'esempio, se l'esecuzione è seriale o parallela, il risultato è differente: l'operazione terminale *count()* decide che non ha bisogno di eseguire le operazioni map e ricava direttamente il conteggio.

```
long cnt = IntStream.range(1, 20)
  // .parallel()
  .map((i) -> {
  System.out.println(i);
  return i * 2;
}).limit(5).count();
System.out.println(">" + cnt);
```

Ovviamente, l'effetto collaterale della stampa dell'indice corrente è illegale, quindi non abbiamo nessuna garanzia sulla sua esecuzione.
La pipeline può, in qualsiasi momento, essere eseguita in modo sequenziale o parallelo semplicemente configurata come tale, tramite l'estensione di un'interfaccia:

```
/**
 * Base interface for streams, which are sequences
 * of elements supporting sequential and parallel
 * aggregate operations.
 */
interface BaseStream< T,S extends BaseStream< T,S > >
```

Segue quindi la configurazione come parallelo:

```
/**
 * Returns an equivalent stream that is parallel.
 * May return itself, either because the stream
 * was already parallel, or because the underlying
 * stream state was modified to be parallel.
 */
S parallel()
```

```
/**
 * Returns an equivalent stream that is sequential.
 * May return itself, either because the stream
 * was already sequential, or because the underlying
 * stream state was modified to be sequential.
 */
S sequential()
```

e invece per renderlo sequenziale:

Attenzione: l'ultimo elemento lungo la catena vince: non si possono costruire pipeline di concorrenza mista.
O meglio, è così attualmente. Un aggiornamento della libreria potrebbe cambiare le cose.
(quindi se metto per primo parallel() ma poi metto sequential(), prevale quest'ultimo)

Una pipeline *ORDERED* garantisce di ritornare gli elementi nell'ordine di emissione, con i passi intermedi che mantengono questa caratteristica senza modifica dell'ordine. Si può notare come il lavoro viene distribuito a più thread, ma il risultato finale sia comunque ordinato:

```
IntStream.range(1, 20).parallel().map((i) -> {
  System.out.println(i + " " +
    Thread.currentThread().getName());
  return i * 2;
}).forEachOrdered((i) -> {
  System.out.println(">>> " + i);
});
```

Una pipeline *SORTED* mantiene gli elementi ordinati secondo il loro ordinamento naturale o dato da un Comparator. *DISTINCT* garantisce che non ci siano elementi uguali secondo equals(), *SIZED* ha la pipeline di dimensione nota (se è anche *SUBSIZED* dice che ci possono essere sottonsiemi efficientemente dimensionati per esecuzione parallela), con la maggior parte delle operazioni che sono *non invasive* (non interferendo con elementi delo stream, cioè non modificano e/o interferiscono con gli elementi dello Stream) e prive di stato interno (*funzioni pure*).

```
/**
 * Returns whether any elements of this stream match
 * the provided predicate. May not evaluate the
 * predicate on all elements if not necessary for
 * determining the result.
 */
boolean anyMatch(Predicate< ? super T > predicate)
```

Altre operazioni sono *short-circuiting,* quindi possono interrompere l'esecuzione della pipeline prima di tutti gli elementi. Esaminiamo quindi il match degli elementi presenti con analisi del risultato:

Il comportamento di findAny() è pensato per la massima performance negli stream paralleli. Ha un comportamento non deterministico; trova un elemento in maniera casuale, quindi velocemente.

```
/**
 * Returns an Optional describing some element of the
 * stream, or an empty Optional if the stream is empty.
 * The behavior of this operation is explicitly
 * nondeterministic; it is free to select any element
 * in the stream.
 */
Optional< T > findAny()
```

```
public class CandidateNumber {
  public final int n;
  public final List< Integer > divisors;

  CandidateNumber(int n, List< Integer > divisors) {
    this.n = n;
    this.divisors = divisors;
  }
}
```

Abbiamo quindi una struttura dati per verificare se un numero può essere candidato ad essere numero primo.

Questa funzione produce, da un intero, un *CandidateNumber* con tutti i suoi divisori che può così essere controllato:

```
public class Divisors implements
  Function< Integer, CandidateNumber > {

  @Override
  public CandidateNumber apply(Integer n) {

    ...

    return new CandidateNumber(n, divs);
  }
}
```

```
public class Perfect implements Predicate< CandidateNumber > {

  @Override
  public boolean test(CandidateNumber c) {
    Integer sum = c.divisors.stream()
        .collect(Collectors.summingInt((Integer n) -> n));
    return (sum + 1) == c.n;
  }

}
```

Questa classe è un predicato, cioè una funzione booleana su un oggetto. Ci dice quando un CandidateNumber è un numero perfetto:

Otteniamo così una ricerca parallela che si ferma al primo risultato:

```
List< CandidateNumber > match = IntStream.range(30, 100000)
  .boxed().parallel().map(new Divisors())
  .filter(new Perfect()).findAny().stream()
  .collect(Collectors.toList());

match.forEach(x -> { System.out.println(x); });
```

Continuazione lezione 12 ed inizio lezione 13: Esempi svolti pt.3

Altre operazioni sono dette *stateful*: può necessitare di consumare tutto o gran parte dell'input per poter emettere l'output mantenendo le caratteristiche desiderate (questo può rendere alcune operazioni apparentemente semplici in realtà molto costose).

Attenzione che limit è stateful perché deve ritornare i primi *maxSize* elementi richiesti.

```
/**
 * Returns a stream consisting of the elements of this
 * stream, truncated to be no longer than maxSize in
 * length.
 */
Stream< T > limit(long maxSize)
```

```
IntStream.range(10, 10000).boxed().parallel()
  .map(new Divisors()).filter(new Perfect()).limit(2)
  .forEach((CandidateNumber c) -> {
    System.out.println(">>> " + c.toString());
  });
```

Vengono ritornati i primi due elementi, in ordine. E non viene ritornato 8128 (calcolo dei primi qui).

Per supportare la parallelizzazione, non basta *Iterator* o *Supplier* che esplicitano le caratteristiche della sorgente, in varie sezioni indipendenti che possono essere elaborate indipendentemente, come *Spliterator*.

```
/**
 * An object for traversing and partitioning elements
 * of a source.
 *
 */
public interface Spliterator< T >
```

```
/**
 * If a remaining element exists, performs the given
 * action on it, returning true; else returns false.
 *
 */
boolean tryAdvance(Consumer< ? super T > action)
```

Qui il metodo di avanzamento è *tryAdvance()*, che ribalta il funzionamento perché non è l'utilizzatore che ottiene il nuovo elemento ma è lo stream che fornisce l'elemento al codice che ci deve operare.

Esso aumenta l'espressività di Iterator, fornendo una stima gli elementi rimanenti, esplicitando le caratteristiche della sorgente, attraversando tutti gli elementi in massa e suddividendo l'iterazione in vari rami (manca invece, coerentemente con la definizione di Stream, l'operazione di rimozione di un elemento). Vediamo una serie di metodi supportati ed esplicativi nei commenti del codice, ad esempio *estimateSize()*:

```
/**
 * Returns an estimate of the number of elements that
 * would be encountered by a forEachRemaining()
 * traversal, or returns Long.MAX_VALUE if infinite,
 * unknown, or too expensive to compute.
 *
 * @return the estimated size
 */
long estimateSize()
```

```
/**
 * Returns a set of characteristics of this
 * Spliterator and its elements.
 *
 * @return a representation of characteristics
 */
int characteristics()
```

Da notare qui il tipo di ritorno. Ci si attende che ogni chiamata a questo metodo dia lo stesso risultato; eventualmente, il risultato può cambiare dopo la separazione.

L'implementazione di default chiama tryAdvance per ogni elemento (similmente esiste *trySplit* per verificare se alcuni elementi possono essere divisi:

```
/**
 * Performs the given action for each remaining element,
 * sequentially in the current thread, until all elements
 * have been processed or the action throws an exception.
 *
 */
default void forEachRemaining(Consumer< ? super T > action)
```

```
/**
 * Performs a reduction on the elements of this stream,
 * using the provided identity value and an associative
 * accumulation function, and returns the reduced value.
 *
 * @param identity the identity value for the accumulati
 *                 function
 * @param accumulator an associative, non-interfering,
 *                    stateless function for combining
 *                    two values
 * @result the result of the reduction
 *
 */
T reduce(T identity, BinaryOperator< T > accumulator)
```

Come conseguenza, lo stream può essere usato in maniera efficace, pianificando tutta l'esecuzione, tra cui anche la quantità di parallelismo adottabile. Le operazioni sono un po' differenti, infatti:

Vediamo poi un esempio pratico, con la riduzione di un range di interi su uno stream avviato come parallelo e capendo il thread in esecuzione, prendendone il nome. Ritorna una somma sul numero di elementi presenti e stampa il risultato.

```
int res = IntStream.range(1, 1001).parallel().
  reduce(0, (a, b) -> {
    System.out.println(a + "+" + b + "=" + (a + b)
      + " " + Thread.currentThread().getName());
    return a + b;
  });
System.out.println(">>>> " + res);
```

```
/**
 * Performs a mutable reduction operation on the elements
 * of this stream using a Collector.
 *
 * @R the type of the result
 * @A the intermediate accumulation type of the Collector
 * @param the Collector describing the reduction
 * @result the result of the reduction
 */
< R,A > R collect(Collector< ? super T,A,R > collector)
```

Si produce una serie di oggetti, avendo poi ForkJoinPool e main che rimangono attivi in questa esecuzione. Siccome la velocità di consumo degli oggetti può essere più o meno veloce, si può implementare un oggetto mutabile così conservando gli oggetti per un certo risultato, o meglio li *colleziona*.

Vediamo poi l'esempio pratico: collezione di una somma di numeri eseguita in parallelo, collezionando il risultato mutabile e stampandolo.

```
int res = IntStream.range(1, 1001).boxed().parallel()
  .collect(Collectors.summingInt((i) -> i));
System.out.println(">>>> " + res);
```

In dettaglio poi si vede *Collector*, discutendo che nell'operazione di riduzione gestita da *reduce* l'accumulazione del risultato avviene creando nuovi valori ma può non essere efficiente.

```
String res = IntStream.range(1, 1001).boxed()
  .map((i) -> i.toString())
  .parallel().reduce("", String::concat);
System.out.println(">>>> " + res);
```

Qui ogni passo di riduzione produce una nuova stringa, sempre più grande, creando una forte pressione per il Garbage Collector.

Collector gestisce una riduzione dell'accumulatore come oggetto mutabile per ragioni di efficienza.

```
/**
 * A mutable reduction operation that accumulates input
 * elements into a mutable result container.
 *
 * @T the type of input elements to the reduction
 *       operation
 * @A the mutable accumulation type of the
 *       reduction operation
 * @R the result type of the reduction operation
 */
interface Collector< T,A,R >
```

La classe quindi produce dei *Collector* a partire dagli elementi di base, un *Supplier<A>* del contenitore del risultato, un *BiConsumer<A, T>* che accumula un elemento, un *BinaryOperator<A>* che combina due contenitori parziali e *Function<A, R>* che dal contenitore ottiene il risultato finale. Con tutti questi mezzi, uno Stream ha tutte le parti della strategia utili per applicare l'algoritmo.

```
String res = IntStream.range(1, 1001).boxed()
  .map((i) -> i.toString()).parallel().collect(
    // supplier
    () -> new StringBuffer(),
    // accumulator
    (acc, el) -> acc.append(el),
    // combiner
    (resA, resB) -> resA.append(resB))
  .toString();
System.out.println(">>>> " + res);
```

Segue un'applicazione classica di quanto appena descritto, chiamato *Template Method* (quindi, uso un fornitore di valore iniziale (*supplier*), accumulatore di risultato intermedio (*accumulator*) e combinazione del risultato in quel momento (*combiner*).

Gli *Stream* sono quindi un'ottima astrazione per modellare collezioni di elementi, modellando in modo semplice algoritmi su insiemi di dati. Normalmente siamo noi a decidere se usare o meno il parallelismo; tuttavia, è lo Stream a decidere autonomamente quanto parallelismo usare, attraverso *ForkJoinPool* (che di standard usa #core-1 threads).

Come possiamo caratterizzare il grado di parallelismo utile?
Viene introdotto il *blocking factor*, quindi l'intensità di calcolo di un algoritmo che indica quanto sia incline a occupare il calcolo (con BF=0 occupa costantemente la CPU (*CPU-bound)*, con BF=1 è costantemente in attesa di I/O (*I/O bound)*). Il calcolo vero è:

$$\#threads <= \frac{\#of\ cores}{1-BF}$$

lavorando meglio a seconda del valore con un numero di thread maggiore/minore del numero di core.
Quindi, se l'algoritmo effettua molta I/O può essere utile modificare l'impostazione standard del ForkJoinPool per aumentare i thread disponibili.

L'uso dello Stream come modello di calcolo può quindi ben semplificare l'aspetto, leggibilità e manutenibilità del codice, come si vede confrontando i due metodi:

```
public static double compute1(int n, int k) {
  int index = n;
  int count = 0;
  double result = 0;

  while(count < k) {
    if(isPrime(index)) {
      result += Math.sqrt(index);
      count++;
    }
    index++;
  }
  return result;
}
```

```
public static double compute2(int n, int k) {
  return Stream.iterate(n, e -> e + 1)
               .filter(Sample::isPrime)
               .limit(k)
               .mapToDouble(Math::sqrt)
               .sum();
}
```

Esempio pratico: TicTacToe - Gioco risolvibile e che esplora tutti i risultati possibili in uno spazio limitato ma non piccolissimo, vedendo se è possibile parallelizzarlo.
Esaminiamo un primo approccio *breadth-first* (in larghezza), creando lo stream delle mosse partendo dalla prima e calcolando le successive. Da questo poi calcoliamo nuovamente tutte le altre, scartando le situazioni conclusive.

```
private static List< Game > breadth(List< Game > games) {
  List< Game > res = new ArrayList<>();
  for (Game g : games) res.addAll(g.moves());
  return res;
}
```

elaborando tutte le possibili mosse a partire dalla posizione e dall'ordine di esecuzione:

```
Game root = Game.setup();
List< Game > first = root.moves();
List< Game > second = breadth(first);
List< Game > third = breadth(second);
List< Game > fourth = breadth(third);
```

A partire dalla quinta mossa, è possibile avere un pareggio (il quale richiede almeno sei mosse) oppure una vittoria. Dobbiamo quindi filtrare le partite terminate da quelle che possono proseguire. Vogliamo quindi introdurre un contatore per tenere conto dei risultati:

```
class CountStatus {
  public final List< Game > games;
  public final int wonO, wonX, ties;

  CountStatus(List< Game > games, int wonO,
      int wonX, int ties) {
    this.games = games;
    this.wonO = wonO;
    this.wonX = wonX;
    this.ties = ties;
  }
}
```

```
private static CountStatus breadth(CountStatus status) {
  int wO = 0, wX = 0, ts = 0;
  List< Game > res = new ArrayList<>();
  for (Game g : breadth(status.games))
    switch (g.status) {
      case PLAYER_O_WON: wO++; break;
      case PLAYER_X_WON: wX++; break;
      case TIE: ts++; break;
      case ONGOING: res.add(g);
    }
  return new CountStatus(res, wO + status.wonO,
      wX + status.wonX, ts + status.ties);
}
```

creando un nuovo metodo che controlla lo stato delle partite, aggiornando il contatore dei pareggi e/o vittorie, e verifica lo stato di gioco dinamicamente all'interno di due strutture di tipo lista. Attenzione: ArrayList non è thread-safe e non deve essere usato in ambiente concorrente se non esplicitamente sincronizzato.

proseguendo l'analisi delle partite dal sesto livello riprocessando l'elemento precedente, accumulando risultati mentre lo spazio delle partite diminuisce:

```
CountStatus fifth = breadth(new CountStatus(fourth));
CountStatus sixth = breadth(fifth);
CountStatus seventh = breadth(sixth);
CountStatus eighth = breadth(seventh);
CountStatus ninth = breadth(eighth);
```

Una partita è lunga al massimo nove mosse; dopo queste tutte le partite sono necessariamente terminate e possiamo stampare i risultati. Di fatto, il gioco è sbilanciato a favore del primo giocatore.
Come si può quindi realizzare una versione della stessa ricerca che sfrutti la concorrenza arrivando al risultato nel minor tempo con l'uso della libreria standard?
Si deve quindi usare uno *Spliterator* che funge da attraversamento per le componenti che calcolano i punteggi e anche i giocatori, gestendo tutto in modo concorrente, senza chiederlo a noi.

```
public class GameIterator implements Spliterator< Game > {

  Queue< Game > current;

  public GameIterator(Game seed) {
    current = new LinkedList< Game >(seed.moves());
  }
```

Teniamo quindi una coda delle posizioni pronte ad essere ritornate, accordando le mosse legali a quella posizione per il consumatore, attraversando un ordine passando poi al successivo, aggiungendo tutte le mosse alla coda nel caso utile.

```
@Override
public boolean tryAdvance(Consumer< ? super Game > action) {
  boolean result = false;
  if (!current.isEmpty()) {
    Game res = current.remove();
    current.addAll(res.moves());
    action.accept(res);
    result = true;
  }
  return result;
}
```

```
@Override
public Spliterator< Game > trySplit() {
  Spliterator< Game > res = null;
  if (current.size() > 1) {
    List< Game > dest = new ArrayList<>();
    int len = current.size() / 2;
    for (int i = 0; i <= len; i++)
      dest.add(current.remove());
    res=new GameIterator(dest.iterator());
  }
  return res;
}
```

Nel caso ci venga richiesto di separare l'iterator, prendiamo metà delle posizioni correnti e le mettiamo nel nuovo iteratore, restituendolo.

Scritto da Gabriel

Manteniamo quindi un nuovo costruttore per un nuovo iteratore partendo da un insieme di elementi:

```
private GameIterator(Iterator< Game > current) {
  this.current = new LinkedList< Game >();
  while (current.hasNext())
    this.current.offer(current.next());
}
```

```
@Override
public long estimateSize() {
  return -1;
}

@Override
public int characteristics() {
  return Spliterator.IMMUTABLE | Spliterator.DISTINCT |
    Spliterator.NONNULL;
}
```

implementando infine una stima della dimensione (non calcolabile) e le caratteristiche dello stream (come visto sopra nella presentazione dei metodi di Spliterator):

Usiamo poi un contenitore per il calcolo delle somme parziali e così eseguire lo stesso calcolo per tutte le mosse:

```
class Score {
  int wonX = 0, wonO = 0, ties = 0;

  Score(int wonX, int wonO, int ties) {
    this.wonO = wonO;
    this.wonX = wonX;
    this.ties = ties;
  }

  static Score ONGOING = new Score(0, 0, 0);
  static Score WON_X = new Score(1, 0, 0);
  static Score WON_O = new Score(0, 1, 0);
  static Score TIE = new Score(0, 0, 1);
```

```
static Score sum(Score a, Score b) {
  return new Score(a.wonX + b.wonX, a.wonO + b.wonO,
    a.ties + b.ties);
}

static Score score(Game game) {
  return switch (game.status()) {
    case PLAYER_O_WON -> Score.WON_O;
    case PLAYER_X_WON -> Score.WON_X;
    case TIE -> Score.TIE;
    case ONGOING -> Score.ONGOING;
  };
}
```

Definiamo poi la somma dei punteggi come il calcolo sul numero di pareggi/vittore/sconfitte e tramite uno switch decretiamo il punteggio, per quanto appena descritto e visto anche sopra.

Costruiamo uno stream che verifica con un iterator il corretto setup del gioco su uno stream parallelo, filtrando i risultati e collezionando solo quelli che sono attivi (*onGoing*), riducendo poi il punteggio attraverso una somma.

```
Score res = StreamSupport
  .stream(new GameIterator(Game.setup()), true)
  .parallel()
  .filter(g -> g.status() != GameStatus.ONGOING)
  .collect(Collectors.reducing(
    new Score(0, 0, 0),
    Score::score, Score::sum)
  );
```

Usiamo infine un “trucco” per sommare i risultati senza passare per un oggetto, sapendo che non superano 2^{20} (memorizzandoli quindi come decimali o similare):

```
long res = StreamSupport
  .stream(new GameIterator(Game.setup()), true)
  .parallel()
  .filter(g -> g.status() != GameStatus.ONGOING)
  .mapToLong( g -> switch (g.status()) {
    case PLAYER_O_WON -> 0x1L;
    case PLAYER_X_WON -> 0x100000L;
    case TIE -> 0x10000000000L;
    case ONGOING -> 0x0L;
  }).sum();
```

```
System.out.println(
  "Won O: %d Won X: %d Ties: %d (%d)"
    .formatted(
      res & 0xfffff,
      (res >> 20) & 0xfffff,
      (res >> 40) & 0xfffff,
      (end - start)));
```

Lezione 14 – Programmazione distribuita

Parliamo per l'appunto di *programmazione distribuita*, quindi teorie e insieme di tecniche che operano in modo coordinato allo svolgimento di un unico compito. È complesso gestire queste tecniche, soprattutto cercando di coordinare processi su macchine differenti.
Le motivazioni che storicamente hanno portato allo sviluppo di algoritmi di questo tipo sono:

- l'*affidabilità*, poter proseguire sui nodi ancora in linea rispetto a quelli fermi/in errore (ARPANET nasce con questo scopo);
- la *suddivisione del carico*, perché la singola macchina non può scalare in maniera indefinita, gestendo bene tutte le risorse. Ciò è possibile se il lavoro lo consente;
- la *diffusione* dei risultati su più nodi, sfruttando la località dei dati per permettere agli utenti che accedono ad un determinato nodo di accedere ai dati più velocemente, anche in assenza di connettività. Ciò era utile già ai tempi delle architetture mainframe.

Le caratteristiche di un algoritmo distribuito sono:

- *concorrenza dei componenti*, con i vari nodi di esecuzione che operano contemporaneamente ma che non condividono risorse;
- *totale asincronia*, ordine temporale di eventi non strettamente condiviso e che deve essere imposto nella maniera giusta (ad es. la sincronizzazione dell'orario, difficile a livello fisico). Non esiste un *global clock* a cui allinearsi;
- *fallimenti imperscrutabili*, quindi i nodi possono fallire/scomparire/comportarsi in modo errato indipendentemente l'uno dall'altro, non distinguendo se l'errore è dovuto al mancato contatto in rete oppure ad un collegamento fisico sbagliato. Individuare il fallimento del nodo è un compito a sé stante.

I nodi comunicano tra di loro tramite *messaggi*, con astrazioni di invio/ricezione dei messaggi e gestione di invio/attesa (con tutte le problematiche del caso di comunicazione).
Uno dei primi tentativi per cercare di facilitare la gestione dei messaggi è stata l'idea di scriverli come chiamata funzionale, mascherando la distinzione fisica tra la macchina chiamante e chiamata. Il termine che identifica questa gestione è *RPC – Remote Procedure Call*, rendendo trasparente la localizzazione del codice chiamato nascondendo i dettagli di come la comunicazione avviene nel client e adattandola ad esso tramite gli *stub* (oggetto con dati predefiniti usato come test).

In particolare, RPC è un protocollo con cui un programma può richiedere un servizio da un programma localizzato in un altro computer nella rete senza conoscere i dettagli della rete stessa. Essa usa trasferimento di parametri e restituzione di valori nelle funzioni, allineando i livelli di elaborazione e attuando una comunicazione bidirezionale I/O.
Una RPC comporta i seguenti passi:

- il client chiama lo stub locale
- lo stub mette i parametri in un messaggio (*marshalling*) e li struttura
- lo stub invia il messaggio al nodo remoto
- sul nodo remoto, un *server stub* attende il messaggio remoto
- il *server stub* estrae i parametri (*unmarshalling*)
- il *server stub* chiama la procedura locale

La risposta seguirà il percorso inverso.

Nei linguaggi orientati agli oggetti, questa procedura è definita come *RMI-Remote Method Invocation*, con l'obiettivo di rendere lo scambio di messaggi equivalente alla chiamata di metodo di un oggetto locale.
Al problema della comunicazione si aggiunge quello dell'indirizzamento dell'oggetto destinatario della chiamata.

Il componente server viene chiamato *Object Broker*, con un primo standard di RMI presente dagli anni 90 come *CORBA (Common Object Request Broker Architecture)*, che fa interagire gli oggetti di tecnologie differenti. Java implementa quindi dalle prime versioni un sistema di RMI e anche di specifica CORBA, partecipando a sistemi distribuiti.

Una chiamata RMI comporta quindi:
- il client chiama lo stub locale
- lo stub prepara i parametri in un messaggio
- lo stub invia il messaggio e il client è bloccato
- il server riceve il messaggio
- il server controlla il messaggio cercando l'oggetto chiamato
- il server recupera i parametri e chiama il metodo destinatario
- l'oggetto esegue il metodo e ritorna il risultato
- il server prepara il risultato in un messaggio
- il server invia il messaggio allo stub
- lo stub riceve il messaggio di ritorno
- lo stub recupera il risultato dal messaggio
- lo stub ritorna al client il risultato

Quante cose possono andare male in questo processo? Alcuni esempi:
- cosa succede se un messaggio non viene recapitato?
- cosa succede se il client e server hanno oggetti con versioni diverse?
- cosa succede se client e server non riescono ad indirizzarsi?
- cosa succede se client/server falliscono?
- come nascondere la complessità?

Gli ambienti di esecuzione moderni hanno quindi problematiche comuni:
- le reti possono essere inaffidabili
- difficile per sistemi di decine/centinaia di nodi essere tutti aggiornati alla stessa versione
- firewall/reti temporanee o wireless rendono impossibile indirizzare un singolo terminale
- i nodi entrano/escono dalla rete con grande facilità ed è facile falliscano

L'evoluzione ha reso poco pratico l'utilizzo di queste tecnologie, sopravvivendo solo in ambienti controllati. Infatti, il modulo *java.corba* è stato ufficialmente deprecato/rimosso in Java 11. RMI è ancora presente ma poco usato nella sua forma più base. I vari strumenti che lo compongono sono in corso di rimozione già da diverse versioni di Java.

Un passo fondamentale evidenziato dai sistemi di RMI è la cosiddetta *serializzazione*, quindi la predisposizione di un oggetto per la trasmissione di messaggi.
Serializzare significa codificare l'oggetto e tradurlo in messaggio applicando il *marshalling*, *deserializzare* è *unmarshalling.* Il problema è molto semplice, ma in realtà risulta essere tema complesso per efficienza/sicurezza. L'enfasi è sul prendere l'oggetto come un tutt'uno e comportarsi in modo il più automatico possibile, in contrasto per es. a collocare i campi dell'oggetto in una struttura predefinita e non dipendente da quest'ultimo. Essa quindi trasforma un oggetto in uno stream di byte.

Il meccanismo di serializzazione nativa a Java è la *java.io.Serializable*, usando l'idea della "*marker interface*" (interfaccia senza metodi o costanti), sfruttando il vantaggio del polimorfismo ma gestendo a mano il cambiamento strutturale delle classi e serializzando grafi di oggetti, indicando quali devono/non devono essere serializzati e assicurando integrità/affidabilità nelle operazioni di marshalling/unmarshalling.
Sono imposte una serie di condizioni (la presenza di un UUID con determinate caratteristiche, restrizioni sui tipi dei campi ecc.) unicamente come "convenzioni", non controllabili dal compilatore.

Il problema della serializzazione è che mittente e ricevente possono ricevere versioni diverse dell'oggetto serializzato e risulta necessario avere un metodo che verifichi integrità/affidabilità dei dati ricevuti, specie se abbiamo oggetti che contengono altri oggetti, che potrebbero potenzialmente non essere rappresentabili.

Nella pratica quindi: *non usare serializzazione nativa di Java per motivi di sicurezza* (es. recente i problemi al framework Spring citati più di una volta dal prof). Inoltre, per il modo in cui funzionano reti e sistemi distribuiti oggi, sono diventati praticabili/preferibili in molte situazioni protocolli testuali trasportati da HTTP e umanamente leggibili. La differenza è alla base della distinzione fra Marshalling e Parsing: una serializzazione usa la prima, un protocollo usa il secondo. I protocolli binari sono quindi riservati ad ambienti con particolari esigenze di efficienza e quindi non parleremo di serializzazione.

Il marshalling nel contesto della programmazione per computer è la trasformazione di componenti archiviati nella memoria del dispositivo in dati utilizzabili che possono essere utilizzati da uno o più programmi che risiedono sul disco rigido. Per quanto concettualmente simili, il marshalling prende i parametri da un punto all'altro, mentre la serializzazione copia dati con una specifica struttura ad una forma primitiva come appunto uno stream di byte. Il marshalling è letteralmente RMI, dunque passare dei dati Object ad oggetti remoti.

Ecco quindi che per far comunicare nodi distribuiti usiamo:

- Sockets (TCP/IP, protocollo affidabile)
- Datagrams (UDP, protocollo non affidabile)

Si vuole quindi spingere nello sviluppo Java delle *fiber*, quindi dei thread leggeri e scalabili (superando il limite numerico di connessioni gestibili dai normali thread, arrivando al milione contro le decine di migliaia gestite dai thread). Si fa riferimento ad esempio a Project Loom, cioè un'implementazione di thread virtuali con le caratteristiche descritte.

java.nio offre una serie di implementazioni asincrone efficienti sulla base del sistema operativo in uso, come l'astrazione *Channel* per unificare le operazioni I/O su canali differenti di esecuzione, *HttpClient* per gestire le richieste http sincrone o meno si usa *HttpClient* o anche la classe URL in modo più semplice.

L'ecosistema Java offre comune librerie robuste ed efficaci atte allo scopo, come ad esempio *Thrift* come RPC scalabile ed è donazione di Facebook, *gRPC* utile come framework open source sviluppato da Google e altri sulla stessa linea. Lo standard JakartaEE introduce un framework di sviluppo applicazioni client/server basate su astrazioni di livello più alto, offrendo tutt'oggi un vasto insieme di soluzioni per vari contesti reali.

Lezione 15 – Primitive di networking

Parliamo quindi delle principali astrazioni di connessione di rete, partendo dal *Socket*, che garantisce un collegamento punto-punto bidirezionale (*Closeable* implementa l'idea *try-with-resources*):

```
package java.net;
/**
* This class implements client sockets (also
* called just "sockets"). A socket is an
* endpoint for communication between two machines.
**/
public class Socket implements Closeable;
```

con l'idea di costruttore costruita nel seguente modo (qui possono essere lanciate eccezioni di sicurezza a causa del numero della porta o dell'indirizzo specificato).
Un client Socket è un socket per iniziare il collegamento verso un'altra macchina.
Un server Socket è un socket per attendere che un'altra macchina ci chiami.

```
/**
* Creates a socket and connects it to the specified
* remote address on the specified remote port.
* The Socket will also bind() to the local address
* and port supplied.
*
* @param address the remote address
* @param port the remote port
* @param localAddr the local address or null for anyLocal
* @param localPort the local port, o zero for arbitrary
**/
public Socket(InetAddress address, int port,
  InetAddress localAddr, int localPort)
    throws IOException
```

```
/**
* Create a server with the specified port, listen backlog,
* and local IP address to bind to.
*
* @param port the local port, o zero for arbitrary
* @param backlog maximum length of the queue of incoming
* connections
* @param bindAddr the local InetAddress the server
* will bind to
**/
public ServerSocket(int port, int backlog,
  InetAddress bindAddr)
    throws IOException
```

L'idea di *ServerSocket* è la creazione di un backlog (coda) per le connessioni in attesa su una certa porta con un indirizzo associato *(binding*).

Da notare che è necessario specificare l'indirizzo cui il socket è collegato; un server può avere più indirizzi IP locali, e si può specificare che sono accettati collegamenti solo per alcuni di essi.
Il Socket, di per sé, rappresenta un collegamento attivo:

- lato client, il socket diventa attivo una volta completato l'handshake TCP/IP
- lato server, il socket viene ritornato quando un collegamento viene ricevuto e completato

Per esempio, il metodo *accept()* blocca finché non viene ricevuta una connessione. Attenzione: il flusso del programma è ora in mano ad un evento esterno.

```
/**
* Listens for a connection to be made to this socket and
* accepts it. The method blocks until a connection is mac
*
**/
public Socket accept() throws IOException
```

Si parla di *socket leak*, qualora troppi thread aprano socket senza liberare le precedenti e si possono avere cattive risposte. Di più al link: http://www.javamonamour.org/2019/09/sockets-leak-case-study.html

Un Socket correttamente inizializzato fornisce *InputStream* e *OutputStream* per trasmettere dati.
Possiamo quindi avere i classici metodi di get() per entrambi (*getOutputStream()* e *getInputStream()*).
Gli stream sono sottoposti a diverse regole, in quanto sono *thread-safe* (permettendo operazioni atomiche una sola alla volta, pena eccezioni), con buffer limitati, pena scarto dei dati in eccesso. Anche la lettura e la scrittura possono bloccare il thread, comunque alcune connessioni possono essere urgenti.
Una volta terminato il loro uso, i Socket vanno chiusi esplicitamente, tramite il metodo *close*(), che può lanciare come fa accept() una IOException.

La comunicazione via Socket deve richiedere una definizione esplicita tra richieste e risposte; dunque, dallo Stream non possiamo sapere se la richiesta è terminata (non è semplice distinguere fra una interruzione della connessione e la sua chiusura regolare, quindi non si può usare come segnale).

```
try (
  ServerSocket serverSocket = new ServerSocket(portNumber);
  Socket socket = serverSocket.accept();
  PrintWriter out = new PrintWriter(
    socket.getOutputStream(), true);
  BufferedReader in = new BufferedReader(
  new InputStreamReader(socket.getInputStream()));
)
```

Dichiariamo le risorse di cui abbiamo bisogno usando la sintassi try-with-resources. Tutte queste risorse implementano AutoCloseable e verranno automaticamente chiuse all'uscita dal blocco try.

Notate che entriamo nel blocco solo dopo aver ricevuto dati dal socket: la chiamata *Socket.accept()* nella dichiarazione delle risorse è bloccante, e quando arriviamo qui in realtà è sufficiente leggere dallo stream quanto ricevuto. Uscendo dal blocco, tutte le risorse, socket compreso, vengono rilasciate.

```
{
  String inputLine;
  System.out.println("Received data.");
  while ((inputLine = in.readLine()) != null) {
    System.out.println("Received: " + inputLine);
    out.println("Hello " + inputLine);
  }
  System.out.println("Server closing.");
}
```

```
try (
  Socket socket = new Socket("127.0.0.1", portNumber);
  PrintWriter out = new PrintWriter(
  socket.getOutputStream(), true);
  BufferedReader in = new BufferedReader(
  new InputStreamReader(socket.getInputStream()));
) {
  System.out.println("Connected, sending " + args[0]);
  out.println(args[0]);
  System.out.println("Got back: " + in.readLine());
}
```

Anche qui costruiamo le risorse per poi usarle. La chiamata bloccante è il *socket.getOutputStream()* che richiede di collegare il client socket all'indirizzo indicato. Dopodiché, all'interno del blocco implementiamo il protocollo: invio di una riga terminata da \n, e ricezione di una riga allo stesso modo.

Proseguiamo con i *Datagrams*, astrazioni di invio pacchetto singolo verso una o più destinazioni in modo preciso/persistente, verso una/più destinazione/i, non avendo un collegamento fisso (*connectionless*):

```
package java.net;
/**
* Datagram packets are used to implement a connectionless
* packet delivery service.
**/
public final class DatagramPacket
```

Qui non si ha garanzia nell'ordine di arrivo/esecuzione, possibilmente ricevendo eccezioni.
L'idea di ricezione quindi può essere data dai due successivi costruttori (qui length deve essere <= a buf.length):

```
/**
* Constructs a DatagramPacket for receiving packets.
*
* @param buf buffer for holding the incoming datagram
* @param length the number of bytes to read.
**/
public DatagramPacket(byte[] buf, int length)
```

```
/**
* Constructs a DatagramPacket for receiving packets.
*
* @param buf the packet data
* @param length the packet length
* @param address the destination address
* @param the destination port number
**/
public DatagramPacket(byte[] buf, int length,
  InetAddress address, int port)
```

Esso è sempre un pacchetto UDP e ha dimensione massima di 64 KB e, se troppo grande, il sistema operativo dovrà frammentarselo, inviando singolarmente ciascuna delle parti. La condizione di frammentazione è data dalla *MTU (Maximum Trasmission Unit)*, dipendente dal protocollo di rete utilizzato come si può vedere qui a fianco.

Protocollo	MTU (bytes)
IPv4 (link)	68
IPv4 (host)	576
IPv4 (Ethernet)	1500
IPv6	1280
802.11	2304

```
package java.net;
/**
* Constructs a datagram socket and binds it to the
* specified port on the local host machine.
*
* @param port the port to use
**/
public DatagramSocket(int port) throws SocketException
```

Per inviare/ricevere abbiamo una sola classe, senza distinzione di operatività:

Possiamo quindi connettere una *DatagramSocket* già creata ad un indirizzo. In successione abbiamo una serie di metodi, per esempio *connect* che si avvia un indirizzo e su una porta:

```
/**
* Connects the socket to a remote address for this
* socket. When a socket is connected to a remote address,
* packets may only be sent to or received from that
* address. By default a datagram socket is not connected.
*
* @param address the remote address for the socket
* @param port the remote port for the socket.
**/
public void connect(InetAddress address, int port)
```

Non è un controllo di sicurezza: semplicemente l'invio ad un indirizzo diverso è un errore, ed un pacchetto ricevuto da un indirizzo diverso viene scartato. Non vanno chiusi perché implementano AutoClosable.

```
/**
* Sends a datagram packet from this socket. The
* DatagramPacket includes information indicating the data
* to be sent, its length, the IP address of the remote hos
* and the port number on the remote host.
*
* @param p the DatagramPacket to be sent
**/
public void send(DatagramPacket p) throws IOException
```

Si ha il successivo invio remoto di un DatagramPacket (che contiene tutte le info di invio, comprese lunghezza ed IP di riferimento):

nonché impostata ricezione (blocca fino alla ricezione del messaggio. Se il messaggio è più lungo del buffer, viene troncato):

```
/**
* Receives a datagram packet from this socket. When this
* method returns, the DatagramPacket's buffer is filled
* with the data received. The datagram packet also
* contains the sender's IP address, and the port number
* on the sender's machine.
*
* @param p the DatagramPacket to be sent
**/
public void receive(DatagramPacket p) throws IOException
```

```
/**
* Closes this datagram socket.
*
* Any thread currently blocked in receive() upon this
* socket will throw a SocketException.
*
* @param p the DatagramPacket to be sent
**/
public void close()
```

con chiusura del datagram socket (come tutte le risorse di questo tipo, va chiusa in quanto occupa risorse di sistema operativo):

Con i Datagram, la logica del protocollo è differente, perché la dimensione del messaggio è nota (con informazione di ricezione completa) ed è possibile inviare messaggi a più indirizzi contemporaneamente (multicast). Il singolo Datagram è inoltre isolato (quindi non è necessario introdurre nel protocollo dei separatori fra messaggi diversi). Rispetto ai Socket perdiamo:

- affidabilità, quindi garanzia di consegna
- reciprocità, una sola direzione e la risposta richiede di mettersi in ascolto
- dimensione, messaggi lunghi subiscono forte penalità di affidabilità

L'esecuzione, con creazione del datagramma e successiva ricezione viene definita ad esempio come a lato.
Usando un Runnable è infatti possibile avviare una connessione, impostandone invio e ricezione e soprattutto ripeterne l'esecuzione.

```
public void run() {
  byte[] buf = new byte[256];
  DatagramPacket packet = new DatagramPacket(buf, buf.length);
  System.out.println("Server setup. Receiving...");
  try {
    socket.receive(packet);
    String received = new String(
    packet.getData(), 0, packet.getLength());
    System.out.println("Received: " + received);
  } catch (IOException e) { e.printStackTrace();
  } finally { socket.close(); }
}
```

```
DatagramSocket socket = new DatagramSocket();
byte[] buf = args[0].getBytes();
InetAddress address = InetAddress.getByName("localhost");
DatagramPacket packet = new DatagramPacket(
  buf, buf.length, address, PORT);
socket.send(packet);
socket.close();
```

Un client, per contro, si definisce con la creazione di un socket e il buffer con i suoi parametri. Questo è molto più semplice, anche perché non deve attendere una risposta.

Conclusione lezione 15: Socket e Datagram (da URL in poi)

```
package java.net;
/**
* Class URL represents a Uniform Resource Locator, a
* pointer to a "resource" on the World Wide Web.
**/
public final class URL
```

Continuiamo parlando di una classe per interagire con risorse Web, quindi la classe di interazione con il mondo Internet, quindi la classe *URL* (classe non più di pratico utilizzo oggi).

Uno URL ha un formato complesso in grado di esprimere molte cose (protocolli, porte richieste, jar, ecc.). Esse normalmente lanciano una *MalformedURLException*.

```
/**
* Creates a URL object from the String representation.
**/
public URL(String spec) throws MalformedURLException
```

```
/**
* Opens a connection to this URL and returns an
* InputStream for reading from that connection.
**/
public InputStream openStream() throws IOException
```

Da una URL possiamo ottenere direttamente lo stream tramite la richiesta GET, lanciando se servisse eccezione.

Abbiamo un esempio di apertura di stream, eseguita aprendo un lettore di stream e leggendo una particolare linea:

```
BufferedReader reader = new BufferedReader(
new InputStreamReader(
  new URL("https://httpbin.org/get").openStream()));
String line;
while ((line = reader.readLine()) != null) {
  System.out.println(line);
}
```

```
/**
* Returns a URLConnection instance that represents a
* connection to the remote object referred to by the URL.
**/
public URLConnection openConnection() throws IOException
```

Per effettuare una POST dobbiamo richiederlo esplicitamente nella *connection* della URL, segnalando che vogliamo ottenere la *connection* dalla URL per scopi di output:

```
public abstract class URLConnection;
```

(superclasse di comunicazione tra l'applicazione e uno URL)

La chiusura di questo stream segnala che abbiamo completato la costruzione della richiesta. Non significa necessariamente che siano stati inviati i byte scritti finora, o che comincino ad essere inviati solo ora. Esso viene poi recuperato con *getOutputStream()*.

```
/**
* Sets the value of the doOutput field for this
* URLConnection to the specified value.
**/
public void setDoOutput(boolean dooutput)
```

Scritto da Gabriel

Vediamo poi la stampa tramite la classe URL, dando in output nel seguente esempio, la specifica modalità HTTP di utilizzo, con l'interpretazione tramite "form" valorizzato con l'input:

```
URL url = new URL(https://httpbin.org/post);
URLConnection connection = url.openConnection();
connection.setDoOutput(true);

PrintWriter writer = new PrintWriter(
  connection.getOutputStream());
writer.println("test=val");
writer.close();
```

```
BufferedReader reader = new BufferedReader(
  new InputStreamReader(connection.getInputStream()));
String line;
while ((line = reader.readLine()) != null) {
  System.out.println(line);
}
```

Partendo da Java 11, il controllo sulle richieste avviene tramite *HttpClient*, classe più versatile e in grado di fornire supporto alle richieste HTTP. Esso usa il *Builder Pattern*, quindi concatena chiamate a metodi che ritornano un oggetto la cui configurazione viene via via completata, per poi concludersi con l'ultimo metodo che ottiene l'oggetto completamente configurato. Risulta essere un sistema comodo per gestire oggetti che hanno un insieme complesso di parametri e poco elegante da gestire con un insieme di costruttori.

L'idea quindi è stata di gestire la banda tale da garantire il controllo su tutte le risorse di una pagina, in un momento in cui HTTP è in grado di gestire diverse decine di oggetti contemporaneamente:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
  .uri(URI.create("https://httpbin.org/get"))
  .build();
```

Una volta costruita la richiesta, si può eseguire per ottenere una risposta; in questo modo può gestire più richieste in modo efficiente, avendo maggiore controllo e visibilità sulle stesse. Un *builder* non ancora completato può essere usato più volte, copiato per essere ulteriormente modificato, e così via (in modo simile alla pipeline di uno stream).
La risposta si ottiene con un oggetto in grado di gestirla (*handler*):

```
HttpResponse< String > response =
  client.send(request, BodyHandlers.ofString());

System.out.println(response.statusCode());
System.out.println(response.body());
```

Il parametro *BodyHandler* permette di gestire come trattare il corpo della risposta e la classe *BodyHandlers* contiene alcune strategie comuni. Una richiesta può essere anche inviata in modo asincrono. Si ottiene un *CompletableFuture*, versione di Future che accetta istruzioni da eseguire al completamento del calcolo.

```
client
  .sendAsync(request, BodyHandlers.ofString())
  .thenApply(HttpResponse::body)
  .thenAccept(System.out::println);
```

Il metodo HTTP da usare viene quindi definito nella costruzione della richiesta:

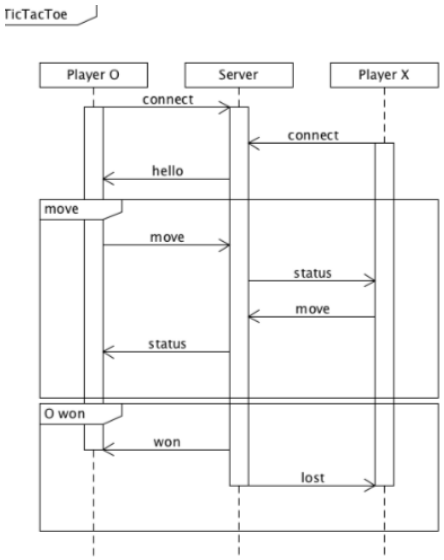
```
HttpRequest delete = HttpRequest.newBuilder()
  .DELETE()
  .uri(URI.create("https://httpbin.org/delete"))
  .build();
```

Usando poi un parametro di tipo *BodyPublisher* per fornire il contenuto di una richiesta POST o PUT:

```
HttpRequest post = HttpRequest.newBuilder()
  .uri(URI.create("https://httpbin.org/post"))
  .timeout(Duration.ofMinutes(2))
  .header("Content-Type", "application/x-www-form-urlencoded")
  .POST(BodyPublishers.ofString("foo=bar&baz=1"))
  .build();
```

A questo punto abbiamo un esempio completo: *TicTacToe*.
Si avrà un client che gioca scegliendo a caso una casella libera e il server risponde alla prima connessione salutando il primo giocatore, facendo lo stesso per la seconda connessione del secondo giocatore.

A quel punto la partita può cominciare e il server richiede la mossa di ciascun giocatore, mostrando il turno e lo stato di gioco, individuando la fine del gioco e chiudendo poi le connessioni.
Il client si collega al server, interpreta la risposta con lo stato della partita ed effettua una mossa a caso tra quelle legali.
L'idea implementativa è quella qui a fianco.




```
try (
Socket socket = new Socket("127.0.0.1",
  ToeServer.PORT_GAME);
PrintWriter out = new PrintWriter(
  socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(
new InputStreamReader(socket.getInputStream()));) {
  System.out.println("Connected.");
  String line;
  boolean done = false;
  while (!done && (line = in.readLine()) != null) {
```

Dopo il setup delle risorse, iniziamo un loop che verrà ripetuto fino a conclusione della partita.

Il protocollo con il server prevede che al giocatore che deve muovere siano presentate uno schema del piano di gioco in tre righe e un elenco di mosse disponibili separato da spazi. Il client usa quest'ultima informazione per scegliere a caso una fra le mosse disponibili.

```
if (line.startsWith("PLAYER")) {
// gestiamo una mossa
  line = in.readLine(); // prima riga
  line = in.readLine(); // seconda riga
  line = in.readLine(); // terza riga
  line = in.readLine(); // mosse disponibili
  String[] split = line.split("\\s");
  String move = split[rnd.nextInt(split.length)];
  out.println(move);
  out.flush();
```

```
  } else if (line.startsWith("Hello")) {
    // partita iniziata
    System.out.println(line);
  } else {
    // partita finita
    done = true;
    System.out.println(line);
  }
```

Gli altri casi del protocollo sono il saluto iniziale che comincia con "Hello", e la conclusione della partita che comincia con un messaggio differente dai due precedenti.

Dichiariamo le risorse necessarie: due socket e due coppie di stream.

```
class GameServer implements Runnable {

int port;
Socket[] sockets = new Socket[2];
PrintWriter[] outs = new PrintWriter[2];
BufferedReader[] ins = new BufferedReader[2];
Game game = new Game();
```

```
try (ServerSocket serverSocket = new ServerSocket(port);) {
  // attendi che i giocatori si colleghino
  connectPlayers(serverSocket);
  // dai al primo giocatore la situazione iniziale
  GameResult status = game.status();
  outs[0].println(status);
  outs[0].flush();
```

Creiamo il collegamento iniziale, attendendo il primo giocatore.

La classe Game guida la selezione del giocatore corrente, agendo sullo stream corrispondente. Si ripete finché la partita non è conclusa.

```
// finché la partita non è conclusa...
while (!status.end) {
  // attendi la mossa dal giocatore
  String move = ins[status.next].readLine();
  // eseguila
  status = game.move(status.next, Integer.parseInt(move));
  if (!status.end) {
    // informa l'altro giocatore
    outs[status.next].println(status);
    outs[status.next].flush();
  }
}
```

```
// comunica il risultato
System.out.println(status);
if (status.valid) {
  outs[status.next].println("You won.");
  outs[(status.next + 1) & 0x1].println("You lost.");
  System.out.println("Player " + (
  status.next == 0 ? "O" : "X") + " won.");
} else {
  outs[0].println("Tied.");
  outs[1].println("Tied.");
  System.out.println("The game is a tie.");
}
```

Alla conclusione, si comunica il risultato ad entrambi gli stream.

chiudendo le risorse una ad una (risulta essere inefficiente e costoso, ma il codice è relativamente semplice ed efficace):

```
// chiudi le risorse
outs[0].close();
outs[1].close();
ins[0].close();
ins[1].close();
sockets[0].close();
sockets[1].close();
```

Lezione 16 – Channels ed inizio Lezione 17 – Fallacies e Frameworks

Per completare la panoramica dei metodi di gestione dei Socket, manca un'astrazione in grado di gestire le richieste di connessione, introdotta a partire da *java.nio* nel 2002 introducendo un albero di tipi dedicato alla gestione della comunicazione nel modo più generico, noto come *Channel*.

```
/**
 * A nexus for I/O operations.
 *
 **/
public interface Channel extends Closeable
```

Esso rappresenta un canale di I/O che può essere aperto o chiuso.

```
/**
 * A channel to a network socket.
 *
 **/
public interface NetworkChannel extends Closeable
```

```
/**
 * An asynchronous channel for stream-oriented
 * listening sockets.
 *
 **/
public abstract class AsynchronousServerSocketChannel
  implements AsynchronousChannel, NetworkChannel
```

Un *NetworkChannel* rappresenta una comunicazione di rete, legata ad un indirizzo (con *bind*) e dichiarando le opzioni supportate.
AsynchronousServerSocketChannel si pone come canale asincrono basato su server socket, accettando in maniera asincrona le connessioni, gestendole.

Per garantire il completamento delle operazioni, abbiamo bisogno di specificare anche questa interfaccia. Essendo la modalità asincrona, questa interfaccia ci permette di indicare al sistema l'azione da effettuare al completamento della successiva interazione.

```
/**
 * A handler for consuming the result of an asynchronous
 * I/O operation.
 *
 * @param V The result type of the I/O operation
 * @param A The type of the object attached to the
 *   I/O operation
 **/
interface CompletionHandler< V,A >
```

```
/**
 * Invoked when an operation has completed.
 *
 * @param result The result of the I/O operation
 * @param attachment The type of the object attached to
 *   the I/O operation when it was initiated
 **/
void completed(V result, A attachment)
```

CompletionHandler ci permette di implementare un oggetto che gestisce la ricezione di un'operazione I/O asincrona, indicando il metodo *completed* per segnalare cosa succede al completamento di una certa operazione e predisponendo eventualmente l'operazione successiva:

Similmente abbiamo il metodo *failed* per gestire l'eccezione incontrata in una certa interazione:

```
/**
 * Invoked when an operation fails.
 *
 * @param exc The exception to indicate why the I/O
 *   operation failed
 * @param attachment The type of the object attached to
 *   the I/O operation when it was initiated
 **/
void failed(Throwable exc, A attachment)
```

Con l'implementazione di un *CompletionHandler* possiamo esprimere il comportamento del server alla prossima interazione, in maniera asincrona (nel senso non sincronizzata con il nostro codice, infatti indichiamo al sistema quale codice eseguire quando si completa un certo evento) ed il parametro *attachment* permette di far circolare le informazioni di contesto in merito allo stato di una conversazione (indicando che codice eseguire dopo un certo evento).

```
/**
 * (from AsynchronousServerSocketChannel)
 * Accepts a connection.
 *
 * @param A The type of the attachment
 * @param attachment The object to attach to the I/O
 *   operation; can be null
 * @param handler The handler for consuming the result
 **/
public abstract < A > void accept(A attachment,
  CompletionHandler< AsynchronousSocketChannel,? super A >
  handler)
```

I vari handler potrebbero essere chiamati da Thread diversi in momenti imprevedibili e, da qui, la necessità di dover gestire in maniera esplicita il cambio di contesto, perché le varie linee di esecuzione non avrebbero altrimenti modo di scambiarsi dati correttamente.
La gestione delle operazioni di I/O richiede quindi di specificare sempre l'attachment da far circolare, assieme all'apposito *CompletionHandler* di gestione di completamento.

Con i successivi metodi di lettura e scrittura, gestendo molte connessioni:

```
/**
 * (from AsynchronousSocketChannel)
 * Reads a sequence of bytes from this channel into the give
 * buffer.
 *
 * @param A The type of the attachment
 * @param dst The buffer into which bytes are to be
 *   transferred
 * @param attachment The object to attach to the I/O op.
 * @param handler The completion handler
 **/
public final < A > void read(ByteBuffer dst, A attachment,
  CompletionHandler< Integer,? super A > handler)
```

```
/**
 * (from AsynchronousSocketChannel)
 * Writes a sequence of bytes to this channel from the giver
 * buffer.
 *
 * @param A The type of the attachment
 * @param src The buffer from which bytes are to be
 *   retrieved
 * @param attachment The object to attach to the I/O op.
 * @param handler The completion handler object
 **/
public final < A > void write(ByteBuffer src, A attachment,
  CompletionHandler< Integer,? super A > handler)
```

Un alternativa all'uso di un *CompletionHandler* è l'utilizzo di una versione di questi metodi che ritorna un Future, per esempio di un Socket, con accettazione *(accept())*, lettura *(read())* e scrittura *(write())*, ciascuna con un Future che si adatta al tipo usato e che prende un parametro di tipo *ByteBuffer.*
L'approccio è duale al precedente ed il contesto è dato dal blocco in cui viene eseguito e gestito il *Future*.
La principale differenza è che, se il blocco di codice è unico per tutta la conversazione, il thread gestore è unico e rimane allocato per tutta la conversazione.

```
ExecutorService pool = Executors.newFixedThreadPool(4);
AsynchronousChannelGroup group=
  AsynchronousChannelGroup.withThreadPool(pool);
AsynchronousServerSocketChannel serverSocket =
  AsynchronousServerSocketChannel.open()
  .bind(new InetSocketAddress("127.0.0.1", GAME_PORT), 16);

pool.submit(() -> {
  serverSocket.accept(
    new GameAttachment(1, new Game(), serverSocket, group),
    new AcceptPlayerO());
});
```

Il main prepara le risorse e istanzia l'attachment vuoto per iniziare alla ricezione della prima connessione. La prossima operazione è *AcceptPlayerO*. Notate che possiamo scegliere noi il tipo di *ExecutorService* che il sistema deve usare.

AcceptPlayerO viene richiamata alla ricezione della prima connessione. Annotiamo il socket collegato nell'attachment, e programmiamo per la prossima azione *WriteFirstStatus*.

```
@Override
public void completed(AsynchronousSocketChannel result,
  GameAttachment attachment) {
  System.out.println(Thread.currentThread().getName() +
    " : game " + attachment.id + " connected player O");
  attachment.server.accept(attachment.playerO(result),
    new WriteFirstStatus());
}
```

```
public void completed(AsynchronousSocketChannel result,
  GameAttachment attachment) {

  attachment = attachment.playerX(result);
  GameResult status = attachment.game.status();
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  byte[] bytes = (status.toString() + "\n").getBytes();
  socket.write(wrap(bytes), attachment, new ReadPlayer());
```

Annotiamo la seconda connessione, inviamo lo stato sulla prima e leggiamo la mossa del primo giocatore.

Se abbiamo raggiunto il numero desiderato di partite, segnaliamo al gruppo del canale di cominciare a considerare la chiusura del sistema. Altrimenti predisponiamo, alla ricezione di una nuova connessione, l'apertura di una nuova partita.

```
  // more games?
  if (attachment.id <= 5) {
    attachment.server.accept(new GameAttachment(attachment.id
      new Game(), attachment.server), new AcceptPlayerO());
  } else {
    attachment.group.shutdown();
  }
}
```

Ci mettiamo in attesa della mossa dalla connessione del giocatore che deve muovere, e programmiamo come risposta l'invio dello stato all'altro giocatore.

```
public void completed(Integer result,
    GameAttachment attachment) {
  GameResult status = attachment.game.status();
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  attachment.readBuf.clear();
  socket.read(attachment.readBuf, attachment,
    new WriteStatus());
}
```

```
String input = new String(attachment.readBuf.array(),
  0, result).trim();
Integer move = Integer.parseInt(input);
GameResult initial = attachment.game.status();
GameResult status =
  attachment.game.move(initial.next, move);
```

Leggiamo i dati di input e calcoliamo lo stato dopo la mossa ricevuta.

Avendo infine l'interpretazione degli stati di gioco tali che (le immagini seguono l'ordine dei punti):

1) se la partita non è ancora terminata, prepariamo il messaggio per il prossimo giocatore, e impostiamo la prossima azione su *ReadPlayer*.

```
if (!status.end) {
  // the game goes on
  AsynchronousSocketChannel socket =
    attachment.players[status.next];
  byte[] bytes = (status.toString() + "\n").getBytes();
  socket.write(wrap(bytes), attachment, new ReadPlayer());
```

2) oppure, se abbiamo un vincitore, predisponiamo entrambi i messaggi (il sistema li manderà quando avrà un thread a disposizione) e programmiamo la chiusura dei socket una volta completato l'invio

```
} else if (status.valid) {
  attachment.players[status.next].write(
    wrap("You won.".getBytes()), attachment,
    new CloseSocket(status.next));
  int loser = (status.next + 1) & 0x1;
  attachment.players[loser].write(
    wrap("You lost.".getBytes()), attachment,
    new CloseSocket(loser));
```

3) se siamo in uno stato diverso da *valid* e siamo alla fine allora abbiamo un pareggio e come prima viene comunicato ad entrambi i giocatori

```
} else {
  // we have a tie
  attachment.players[0].write(
    wrap("Tied.".getBytes()), attachment,
    new CloseSocket(0));
  attachment.players[1].write(
    wrap("Tied.".getBytes()), attachment,
    new CloseSocket(1));
}
```

chiudendo in maniera indicizzata ciascun socket dei vari giocatori, facendo riferimento al loro attachment:

```
try {
  attachment.players[idx].close();
} catch (IOException e) {
  e.printStackTrace();
}
```

Introduciamo ora il concetto di *fallacia* per dire cose che sembrano vere ma non lo sono.
A noi interessano in particolare quelle che sono definite come *8 fallacies of distributed computing*, cioè:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

1) The network is reliable

Fino ad ora ci siamo affidati al fatto che il collegamento tra due programmi è affidabile; ciò non è più valido quando siamo all'interno di una rete distribuita, occupandosi di risolvere gli errori di programmazione.
Si deve quindi tenere conto che le connessioni si interrompano, vengano negate o i messaggi vadano persi.
Va poi bilanciato il rischio/costo di un tale evento con costo della ridondanza/affidabilità.

2) Latency is zero

Definiamo *latenza* il tempo che intercorre tra l'emissione e la ricezione del messaggio. Questo tempo deriva da una caratteristica fisica, normalmente alla velocità della luce, normalmente non superabile. Anche a distanze geografiche, la latenza può essere percepibile (per esempio 30 ms per raggiungere le due coste dell'Atlantico). Pensiamo quindi che il recapito dei messaggi sia immediato, quando in realtà non è così.

3) Bandwidth is infinite

La banda disponibile oggi ormai è sempre maggiore, tuttavia sono cambiati i dati da trasferire, che aumentano in maniera altrettanto più grossa, troppo velocemente. Può essere quindi raggiunto il limite disponibile della banda, dando priorità all'affidabilità della comunicazione piuttosto che all'economia di trasmissione, dato che stiamo trasmettendo più dati. Avremo quindi riduzione della banda utile.

Se il messaggio è grande, va gestito come un insieme di parti singole e dunque bisogna considerare anche questo qualora si crei un protocollo e l'overhead da esso generato, tale che il rapporto influenzi correttamente quanto creato.

4) *The network is secure*

Naturalmente l'aspetto sicurezza è fondamentale; purtroppo non abbiamo modo di sapere a meno di mezzi di autenticazione, autorizzazione ed integrità, la garanzia di confidenza delle informazioni trasmesse.

5) *Topology doesn't change*

Oggi l'ubiquità di accesso ha come conseguenza che un dispositivo sia spesso in movimento, dato che i nodi sono in grado di spostarsi da un ramo di rete ad un altro con estrema facilità. Il grafo di rete è in continua evoluzione e quindi si deve disporre di un indirizzamento non dipende dalla topologia della rete, portando quindi ad un indirizzamento possibilmente sempre valido.

6) *There is one administrator*

Le reti sono estremamente frammentate, sia nella struttura che nella responsabilità di gestione. È sempre più difficile chiedere determinate caratteristiche dalla rete di comunicazione. Difficilmente oggi si possono imporre vincoli sulle connessioni e dunque si deve puntare alla massima semplicità nella struttura e nella creazione dei requisiti di un protocollo di comunicazione.

7) *Transport cost is zero*

Da un punto di vista energetico, il costo del trasporto sta calando, ma decisamente non è zero. Pensiamo anche a tutta la infrastruttura server, dispositivi, elettricità, ecc. Trasferire grandi quantità di dati richiede un forte dispendio energetico. Se lavoriamo in un ambito come IoT, l'economia di funzione del dispositivo è decisamente fondamentale. Dunque, efficienza e sintesi diventano la regola d'oro.

8) *The network is homogeneous*

Le reti oggi non sono più costituite da una sola tecnologia o mezzo di comunicazione, in quanto un messaggio attraversa una varietà di mezzi trasmissivi, mezzi di comunicazione e molti apparati. Come già detto, dunque, il protocollo di comunicazione deve evitare di richiedere caratteristiche di trasporto particolari, affidandosi il più possibile alle funzioni più standard.

In tutta la letteratura le fallacie sono 8; tuttavia, in una recente intervista, Deutsch (autore di questi principi) ne ha elencate 9. La nona in particolare fa riferimento a *Time is ubiquitous*; quindi, che si possa ragionevolmente pensare che le operazioni vengano svolte sempre allo stesso momento; ovviamente in realtà dipende dal contesto, dato che una computazione può avvenire in tempi diversi alla stessa ora o viceversa.

Continuazione lezione 17 – Fallacies of distributed computing

Abbiamo visto come costruire manualmente client e server, tuttavia normalmente si lavora a livello più alto, usando delle astrazioni per semplificare la struttura del codice e sfruttare implementazioni già testate e corrette in merito alla creazione di un client/server in rete.

Per esempio, usiamo *VertX*, in carico ad Eclipse Foundation, atta alla realizzazione di app web e servizi moderni/microscalabili. Esso cerca di essere modulare e compatto nelle dipendenze, event-driven nonché fortemente asincrono, scalabile, open-source ed *unopinionated* (libero nell'implementazione), sfruttando un approccio adatto a vari linguaggi, che non segue lo standard. Concretamente va installato come parte delle dipendenze Maven di un progetto e i pezzi di codice che implementa vengono chiamati *verticles*. La

gestione dei suoi eventi è data dall'*event bus*, attraverso cui ciascuno dei verticles agiscono in maniera asincrona. Buon articolo utile e completo da: https://www.baeldung.com/vertx

Useremo quindi i moduli *vertx-core* e *vertx-web* per implementare il funzionamento del server *TicTacToe* come applicazione web vera e propria, valutando le modifiche al protocollo ed al codice da fare. Dobbiamo quindi pianificare l'API del servizio esposto, attraverso una Web API usabile da browser, sia una JSON usabile da client automatico.

La creazione di una API va ben studiata, in quanto richiede stile e giusta preferenza, dato che deve essere un codice possibilmente creato per comunicare con altri sviluppatori ed atto ad un utilizzo continuativo nel corso del tempo. Questo vale sia per le API web come quella che stiamo per discutere, sia per le librerie o anche le singole classi. È importante avere chiaro in mente che si tratta di una comunicazione fra persone (il compilatore, una volta soddisfatto, è fuori dal gioco): quindi precisione, evidenza dell'intento e aiuto a scegliere la strada più corretta sono caratteristiche di considerare di grande valore.

Per esempio impostiamo:

1) *una Home Page*, quindi una pagina HTML di benvenuto, tramite una *GET* (immaginiamo una risposta in formato *text/html*)
2) *una fase di ingresso al gioco*, tramite una redirect al form di gioco tramite HTML e l'accesso all'url per ottenere lo stato del gioco (JSON), tramite una *POST /game*
3) *la fase di mossa in un gioco*, identificata da *move=x*, un form di gioco (semplice formato HTML) ed un JSON per lo stato del gioco, tramite quindi una *POST /game/{id}*

Utilizziamo due strumenti utili, uno molto conosciuto (*curl*) per il trasferimento di dati tramite URL e un altro (*jq*), processore leggero e flessibile per il formato JSON, per la creazione del nostro API.

```
$ curl -X POST -H "Accept: application/json" \
  http://localhost:8080/game
{"game":"/game/1134141953"}

$ curl -H "Accept: application/json" \
  http://localhost:8080/game/1134141953
```

Ad esempio, l'implementazione l'astrazione che il modulo *vertx-web* ci mette a disposizione è un router che ci consente di associare ad una URL una lambda. Questa può modificare la risposta impostando esito e contenuto del gioco ed il framework si comporta di conseguenza.

Cominciamo dunque a creare gli identificatori dei giocatori, creando un *Optional* per le eventuali partite:

```
class NetGame {
  String[] playerId = new String[2];
  private Optional< Game > game =
    Optional.empty();
```

La gestione del server interno gestisce le partite di rete e i giocatori in formato stringa, con una apposita struttura. Successivamente aspettiamo i giocatori, attraverso una struttura *LinkedBlockingQueue*:

```
class GameServer {
  // active and completed games
  ConcurrentMap< String, NetGame > games =
    new ConcurrentHashMap<>();
  // awaiting players
  BlockingQueue< NetGame > openings =
    new LinkedBlockingQueue<>(16);
```

```
public Optional< GameIndex >
  status(String playerId) {
  var res = Optional.empty();
  if (games.containsKey(playerId) &&
    games.get(playerId).started()) {
    int idx = games.get(playerId).playerId[0]
      .equals(playerId) ? 0 : 1;
    res = Optional.of(new GameIndex(
      idx, games.get(playerId).status()));
  }
  return res;
}
```

Prendiamo poi i singoli giocatori, identificando se ciascuno di questi abbia iniziato e se una partita contiene l'identificatore ad un certo giocatore viene presa come informazione, successivamente restituendo lo stato della computazione e del giocatore stesso:

Poi il server si imposta, creandosi il GameServer, le opzioni di gestione HTTP con server e router e l'handler che gestisce tutti i vari percorsi:

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

  // The Game Server
  GameServer gameServer = new GameServer();

  // The infrastructure parts
  Vertx vertx = Vertx.vertx();
  HttpServerOptions options = new HttpServerOptions()
    .setLogActivity(true);
  HttpServer server = vertx.createHttpServer(options);
  Router router = Router.router(vertx);
  // setup body handling for all routes
  router.route().handler(BodyHandler.create());
```

```
// Browser entry point
router.get("/").produces("text/html").handler(ctx -> {
  ctx.response().end(JOIN_FORM);
});
```

Successivamente otteniamo la risposta, completandola con la stringa che descrive il form HTML di benvenuto:

Le chiamate POST sono configurate in due modi:

1) il primo handler viene usato per prendere il contenuto di un primo client sotto forma di testo HTML
2) il secondo handler viene usato per prendere il contenuto di un primo client sotto forma di JSON

```
// POST /game - want to play
router.post("/game").produces("text/html").handler(ctx -> {
  GameLocation loc = new GameLocation(gameServer.create());
  ctx.response().setStatusCode(302)
    .putHeader("Location", loc.game).end();
});

router.post("/game").produces("application/json")
  .handler(ctx -> {
    GameLocation loc = new GameLocation(
      gameServer.create());
    ctx.response().putHeader("content-type",
      "application/json").end(location.toJson(mapper));
});
```

```
// POST /game/{id} move=x - play move x
router.post("/game/:playerId").produces("text/html")
  .handler(ctx -> {
    String playerId = ctx.request().getParam("playerId");
    boolean open = gameServer.open(playerId);
    if (!open)
      ctx.response().setStatusCode(404).end(GAME_NOT_FOUND);
```

Si prende poi la mossa nel caso HTML, ottenendo dal contesto la richiesta, il parametro effettivo e l'ID del giocatore sotto forma di parametro. Se quella partita esiste si apre la connessione del giocatore, rispondendo 404 qualora non ci sia.

Lo status ritorna un Optional, usando una chiamata Map per trasformare lo stato ottenuto in qualcosa di pronto per essere reinviato; a seconda del tipo di risultato si verifica se siamo o meno nello stato finale, prendendo come identificatore un giocatore specifico, un indice ed uno stato, nonché formattando l'attesa di un giocatore, attendendo poi gli altri eventualmente:

```
String result = gameServer.status(playerId)
  .map((res) -> {
    int idx = res.idx; GameResult status = res.status;
    int move = Integer
      .valueOf(ctx.request().getParam("move"));
    if (idx == status.next && status.valid && !status.end)
      status = gameServer.player(playerId, move)
        .orElse(status);
      return render(playerId, idx, status);
    }).orElse(String.format(WAIT_FOR_ANOTHER, playerId));

    ctx.response().end(result);
});
```

```
router.post("/game/:playerId").produces("application/json")
  .handler(ctx -> {
    // ...
      return renderJson(playerId, idx, status);
    }).orElse(new GameStatus("wait for another")
      .toJson(mapper));

  ctx.response().end(result);
});
```

Nella fase di POST, caso JSON, renderizziamo il risultato dell'attesa dei dati *id_giocatore, indice, stato* e mappando il risultato in forma JSON, segnando il risultato finale:

Prendiamo l'ID del giocatore, reindirizzando il browser con una successiva chiamata GET, validando l'ID del gioco richiesto (a seconda che esista o meno) e si mappa il risultato renderizzandolo a seconda del giocatore attivo, attendendo poi gli altri:

```
router.get("/game/:playerId").produces("text/html")
  .handler(ctx -> {
    String playerId = ctx.request().getParam("playerId");
    boolean open = gameServer.open(playerId);
    if (!open)
      ctx.response().setStatusCode(404).end(GAME_NOT_FOUND);

    String result = gameServer.status(playerId).map(
      (res) -> render(playerId, res.idx, res.status))
      .orElse(String.format(WAIT_FOR_ANOTHER, playerId));
  ctx.response().end(result);
});
```

```
public static void main(String[] args) {
  // ...
  server.requestHandler(router).listen(8080);
}
```

Dopo aver così programmato il router, possiamo dire al server di aver configurato tutto quanto e si mette in ascolto sulla porta 8080 rispondendo a tutte le successive richieste HTTP su quella porta:

Concludiamo con alcune osservazioni di massima sull'esempio discusso:
L'approccio del framework è tipico dei Vert.X è tipico dei framework asincroni, fornendo un ambiente con specifici handler richiamati per ogni singolo evento. Conseguentemente, con un po' di disciplina, è possibile mantenere il codice che interagisce il framework e, con una buona gestione degli handler, garantire la giusta operazione degli handler con i loro side-effects. La struttura dichiarativa costringe ad esternalizzare l'organizzazione del codice, dando a noi in carico la gestione coerente dello stato condiviso.

Perché usare un framework?
Un framework fornisce un ambiente facile da gestire all'interno del quale si affrontano un insieme di casi fondamentali, rendendo sicura ed efficiente una successiva implementazione, in maniera tale che gli utenti ne beneficino in maniera automatica.
Per esempio, per un framework di app web, è utile saper specificare facilmente le rotte a cui rispondere, costruendo agilmente le singole risposte. I dettagli in esso configurabili saranno la sicurezza nel trattamento della comunicazione, la suddivisione delle risorse tra le singole parti del sistema e minori particolari implementativi.

Perché non usare un framework?
Un framework presenta una serie di casi da gestire; nel momento in cui si trova un caso non previsto tra quelli d'uso, la gestione potrebbe essere molto complessa.

Nel caso di framework per app web può essere difficile rispondere a richieste esterne/esotiche, controllando bene la gestione della risposta. Lo stesso potrebbe rendere difficile capire dove sia fallita un'esecuzione, magari a causa di qualche bug, avendo alcune falle di sicurezza a causa di qualche default errato e andando magari anche in sovraccarico di risorse, a causa di una mancata limitazione delle stesse. Come utenti, quindi, l'ordine in cui tutte queste idee e concetti applicativi sono implementati ci riguarda direttamente, a seconda dei costi di cambiamento ed evoluzione della stessa interfaccia, considerando anche i cambiamenti del codice.

Lezione 18 – Stato distribuito

Abbiamo fino ad ora giustificato la necessità di creazione di un sistema distribuito per motivi di affidabilità, suddivisione del carico e distribuzione dei risultati tra i vari nodi e gli utenti.
Le stesse caratteristiche sono desiderabili per un insieme di dati, facendo in modo questi siano sempre disponibili, anche in caso di guasti, con una quantità superiore a quella gestibile da una macchina sola e in generale accessibile da più posizioni geografiche.

I singoli nodi però devono essere allineati, coordinati e concordi tra loro. Tale problema venne proposto molto presto, partendo *dall'RFC 1034-1035*, in cui l'associazione tra nomi ed host veniva mantenuta nel file *hosts.txt*, mantenuto dal NIC (Network Information Center). Oltre al problema della crescita vi era anche il problema dei cambiamenti, cercando di mantenere autonomamente il controllo sugli indirizzi, soprattutto in locale e aggiornandosi nel corso del tempo ai nuovi inseriti da NIC.

Si passò quindi a disegnare una versione distribuita e generalizzata del database, descritto da *RFC-882, RFC-883*, cercando di arrivare ad una struttura funzionante creata in questo modo (ancora oggi sotto forma di DNS):

In esso, ogni nodo è responsabile (*autorevole*) per un ramo di albero, con un client che chiede la risoluzione di un nome al DNS cui appartiene o più vicino, propagando poi la richiesta se non ha la risposta.
Il requisito principale è la suddivisione delle responsabilità attraverso la suddivisione gerarchica dello stato stesso, senza condivisione fra i nodi. Se i dati non si possono distribuire gerarchicamente, dobbiamo organizzare un *consenso* tra i nodi. Ad esempio, se il requisito principale è la resistenza ai fallimenti o un vincolo di prestazioni molto elevate, la replicazione del nodo è obbligata, facendo in modo che un guasto non porti a perdite di dati ed aumenti la capacità di risposta del sistema al crescere del carico.
Il problema del consenso, quindi, è cercare di garantire che tutti i nodi del sistema posseggano la stessa versione di un dato.

Esso viene esposto da Lamport (creatore di LaTeX e studioso della programmazione distribuita) descritto come problema dei *Generali Bizantini*: i generali assediano una città e devono decidere in maniera unanime se attaccare o ritirarsi; un attacco parziale è svantaggioso rispetto ad uno coordinato oppure rispetto ad una ritirata. Essi possono comunicare solo tramite scambio di messaggi, ma non si può sapere a priori se saranno leali, perché potremmo avere messaggi errati o non risposta ai messaggi inviati.

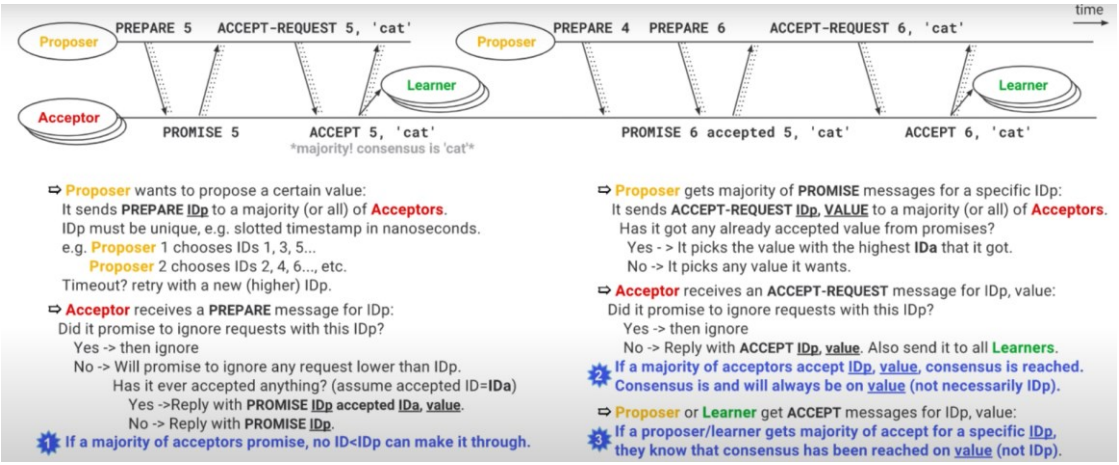
Il problema modella il caso in cui un nodo si comporta in modo diverso ad ogni risposta, non conoscibile all'esterno se ha un comportamento scorretto/nodo guasto.
Per affrontare questo problema, si pone l'uso di un *algoritmo di consenso*, chiamato *PAXOS*, basato su un protocollo a tre fasi che garantisce l'assenza di blocchi nel caso di un guasto singolo; i nodi sono modellati per coprire ogni caso. Esso è l'algoritmo guida di questa casistica.

In questo algoritmo abbiamo: i *proposers*/*proponenti*, che propongono valori per fare ottenere il consenso al leader, inviando la richiesta agli *acceptors*/*Votanti* per raggiungere un Quorum. Una volta che un sufficiente numero di votanti appartiene ad un Quorum, la richiesta è stata accolta.
Gli Ascoltatori, ricevendo messaggi dai Votanti, forniscono ridondanza addizionale in caso di guasti.
Il Leader coordina il protocollo ed è in grado di fermarlo in caso di eccessivi fallimenti in attesa che possa essere ripresa la normale attività.

Scritto da Gabriel

Nella pratica, di solito, i nodi possono ottenere tutti i ruoli; va comunque oltre gli scopi del corso esaminarlo in dettaglio.

Un esempio completo da Google:



Una variante dell'algoritmo permette di fornire garanzie di correttezza anche in presenza di Guasti.
Un algoritmo più recente (2013), chiamato *Raft*, è più comprensibile e suddivide il problema del consenso in sottoproblemi, rendendo il modello semplice ed agevole nell'implementazione.
Il concetto principale è la *replicazione di una macchina a stati*, enfatizzando il ruolo del leader e della sua elezione, garantendo la correttezza dell'algoritmo. Sono disponibili implementazioni in vari linguaggi.
Piccolo esempio step-by-step presente dal sito: http://thesecretlivesofdata.com/raft/

Ogni nodo può rispondere allo stesso modo alla richiesta di un dato; i limiti e compromessi da dover prevedere in caso di guasti è definito dal *CAP Theorem* che sono tre caratteristiche di un DB distribuito:

1) *C (Consistenza)*, ricevendo ad ogni lettura il valore più recente o un errore
2) *A (Availability)*, ogni richiesta riceve una risposta valida (ma non per forza l'ultimo valore)
3) *P (Partition Tolerance)*, il sistema funziona anche se la rete fallisce per un insieme di nodi; quindi, viene partizionata

Il teorema afferma che solo *due* di queste proprietà sono contemporaneamente garantite.
Siccome ogni rete può fallire, in pratica il teorema afferma che: *in caso di partizione di rete un sistema può essere o consistente o disponibile, ma non entrambe le cose.*

Un DB distribuito porta quindi ad una costruzione che, in caso di partizione di rete, garantisca consistenza, avendo che alcune richieste vanno in errore; ciononostante, ognuna di queste ritorna un valore ma non necessariamente l'ultimo.
Un'estensione del teorema, *PACELC*, considera anche il caso della normale operatività:

1) in caso di (P) Partizione, si deve scegliere fra (A) Disponibilità e (C) Consistenza
2) (E) Else/Altrimenti, fra (L) Latenza e (C) Consistenza

Esso è stato sviluppato perché CAP non fa previsioni su performance o latenza; infatti, secondo CAP un DB può essere considerato Disponibile se una query restituisce una risposta dopo 30 giorni, ad esempio.
La maggior parte dei sistemi NoSQL si orienta verso PA/EL (Partizione Disponibilità/Altrimenti Latenza), favorendo disponibilità in caso di partizione e minore latenza nel caso normale, a scapito della consistenza, mentre i relazionali cercano in ogni caso di mantenere la consistenza; quindi, saranno PC/EC (Partizione Consistenza Altrimenti Consistenza).

Finora abbiamo lavorato nell'ipotesi che il sistema distribuito conservi dati che vengono scritti da una sola fonte o che solo occasionalmente arrivino da più di una direzione. Essa non è l'unica scelta, infatti in alcuni sistemi distribuiti è normale che ogni nodo abbia nuove informazioni da fornire e che queste vadano riconciliate e riunite con quelle prodotte indipendentemente da un altro nodo (dove non è significativo decidere quale informazione e chi arriva prima o dopo):

1) sistemi di telepresenza/chat
2) editor collaborativi
3) eventi in real-time con molti partecipanti

Quindi non vi è esigenza di raggiungimento di consenso su un dato, in quanto ogni nodo produce nuove versioni dello stesso dato e deve riunire le modifiche fatte localmente.

In letteratura vi sono *due* soluzioni al problema:

1) Operational Transformation, garantendo che dato un insieme di modifiche, indipendentemente dall'ordine e dal dove avvengano, lo stato finale sia lo stesso in tutti i nodi. Idee come Google Docs/Google Wave non hanno preso piede, dato che non avevano generalità (ogni volta con un oggetto differente bisognava riscrivere da capo un nuovo oggetto)
2) Conflict-Free Replicated Data Type/CRDT *(a volte anche Commutative o Convergent al posto di Conflict*), strutture replicate su più nodi, con la garanzia che si possano riconciliare le modifiche risolvendo ogni possibile conflitto. Esempio banale: un booleano che passa solo da falso a vero. Sono quindi strutture dati tali da evitare conflitti e il cui risultato può essere replicato senza che ci siano problemi a riunire le singole versioni.

Diverse strutture dati implementano CRDT:

1) Grow-only Counter
2) Positive-Negative Counter
3) Grow-only Set
4) 2-Phase Set
5) Last-Write-Wins Set

E anche vari progetti/framework:

- Akka
- Riak DB
- Redis Enterprise
- Facebook Apollo

Un risultato molto recente affronta il problema del consenso distribuito di CAP, proposto in un paper del 2019 proponendo un risultato che caratterizza i programmi e il mantenimento della consistenza in rete. CALM sta per "Consistency As Logical Monotonicity", affermando che i programmi mantengono la proprietà CP nel teorema CAP sono i programmi esprimibili con "*logica monotona*", per cui, aumentandone la dimensione, il risultato non cambia.

Esempio: un sistema distribuito che individua un deadlock al suo interno è logicamente monotono.
Una volta individuato il deadlock, anche con più nodi la soluzione non cambia.
Controesempio: sistema distribuito di Garbage Collection, non è logicamente monotono, perché il sistema cambia con aggiunta e rimozione dei nodi, in particolare per i nodi non più in uso si ha una situazione che deve essere tracciata.

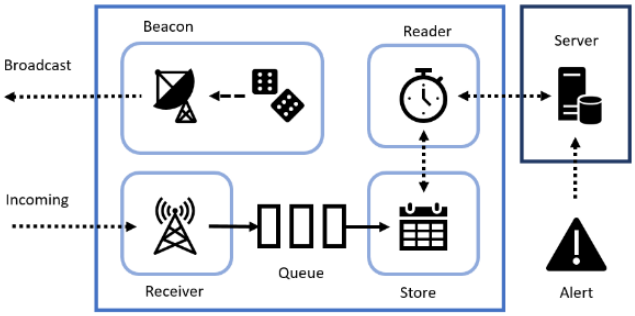
CALM è un risultato costruttivo, perché individua una classe precisa di programmi che forniscono una garanzia, ma tale classe non è molto popolata di programmi. Tuttavia, la logica monotona si pone come strumento di costruzione per ciascuno di questi. Infatti, i CRDT svolgono un ruolo importante nel CALM Theorem, caratterizzando le garanzie di consistenza ottenibili da un sistema.

Lezione 19 - Esempi svolti pt.4

Per concludere la panoramica sui metodi "tradizionali" di costruzione di programmi distribuiti, vediamo un esempio che coinvolge nodi con caratteristiche diverse rispetto al classico richiesta-risposta.
L'esempio riguarda il tracciamento e l'individuazione dei contatti avuti con persone contagiose e prevenire la diffusione della pandemia COVID-19, prevenendo ulteriori contagi.
Il metodo si basa sull'emissione di messaggi crittografati in modo tale da rimanere anonimi, ma che possono essere identificati quando una parte della chiave pubblica viene rivelata. Lo scambio di dati avviene tramite Bluetooth, soddisfacendo i requisiti di tracciamento e, allo stesso tempo, preservare le esigenze di privacy.

In questo modo ogni dispositivo può controllare se una chiave pubblica ha emesso qualche messaggio ricevuto; tuttavia, non sa a chi appartiene ne ottiene altre informazioni. La privacy è inoltre garantita dal fatto che non esiste un deposito principale dei messaggi, essendo i dati gestiti dai dispositivi remoti, ma solo un'autorità centrale che pubblica le chiavi dei contatti rivelatisi positivi.
Questo funzionamento decentralizzato fa in modo che gli attacchi più teorici richiedono risorse considerevoli per essere messi in pratica.

Il protocollo che usiamo è *dp*^{3}*t*, cioè *Decentralized Privacy-Preserving Proximity Tracing*, strutturato vagamente in questo modo:

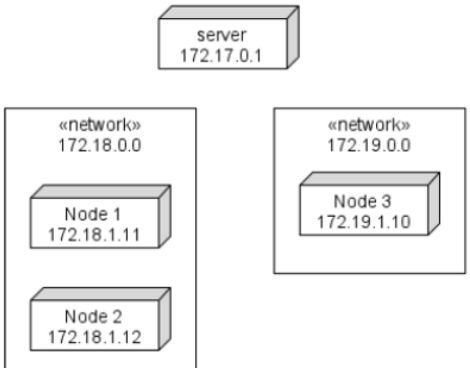


Il sistema è composto da più nodi e ognuno è una persona interessata al tracciamento, ciascuno con componenti indipendenti e concorrenti agli altri, con il server centrale che raccoglie e distribuisce le segnalazioni. Analizziamo i singoli componenti:

1) il *Beacon* emette dei messaggi broadcast indirizzati ad ogni indirizzo della rete a cui appartiene, interagendo ad intervalli regolari messaggi con tutti i nodi in ascolto. Esso dovrà essere un Thread indipendente, con un ciclo che prepara, emette il messaggio ed aspetta un tempo stabilito
2) il *Receiver* ascolta sulla porta indicata, accodando ogni messaggio ricevuto. Esso si pone in ascolto sul DatagramSocket, in maniera tale che quando riceve un messaggio, lo pone sulla coda per la sua elaborazione. Servirà un Thread concorrente, con un flag di chiusura che attende l'arrivo di un messaggio e ne estragga il contenuto.
3) lo *Store* conserva i messaggi ricevuti, controllando se il messaggio segnalato è stato notato in passato. Deve mantenere un insieme di messaggi già visti in modo che possano essere consultati per identificare un contatto, prelevandoli dalla coda e aggiungendoli all'insieme dei messaggi ricevuti. Essa è thread-safe, in modo che il test di appartenenza sia sempre fattibile da parte di un altro thread.
4) il *Reader* interroga ad intervalli regolari il servizio centrale, leggendo i messaggi segnalati e controllando se uno o più di essi sono stati ricevuti. Un timer controlla periodicamente la lista (pubblicata come testo semplice, un messaggio per riga, semplice da capire) da una URL configurata all'avvio; ottenuto l'elenco, viene cercato ogni messaggio nella struttura dati dello Store e ne si verifica la ricezione
5) il server centrale, che pubblica una lista di messaggi segnalati e permette di aggiungere nuovi messaggi alla lista. Esso è realizzato come un servizio web che ad una richiesta GET fornisce i messaggi segnalati come testo e con un POST accetta i nuovi messaggi da considerare segnalati, usando Vertx in maniera thread-safe per le operazioni asincrone.

Per verificare il funzionamento della trasmissione dei messaggi in broadcast, strutturando la rete in modo tale da ottenere il risultato sotto forma di immagine Docker, realizzando i nostri scopi sotto forma di risorse di networking. Il container crede di essere l'unico processo del sistema operativo e ha un suo filesystem, offrendo una totale interazione con le applicazioni esistenti. Esso offre una virtualizzazione in modo più economico e maneggevole rispetto ad una virtualizzazione completa rispetto ad una VM, fornendone però gli stessi vantaggi.

Fondamentalmente si può considerare comodo, perché è come se si predisponesse un intero sistema operativo dedicato ad una particolare applicazione/programma/package con una configurazione immutabile e sempre riutilizzabile ogni qual volta si abbia l'uso in quello specifico contesto.

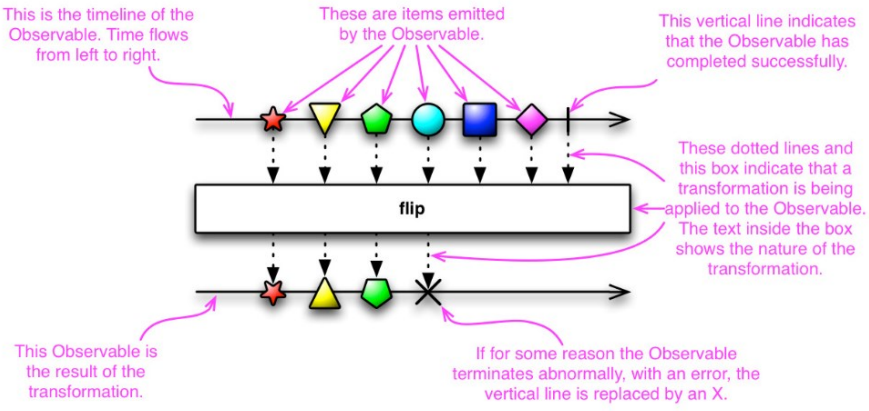


A livello di rete, il funzionamento è il seguente:
- i nodi 1/2 si trovano sulla stessa sottorete
- il nodo tre è in un'altra sottorete
- tutti e tre raggiungono il server allo stesso indirizzo sull'host

Lezione 20 e Lezione 21: Reattività e Reactive Streams

Per cercare astrazioni di livello più alto con cui gestire, in modo coordinato, le problematiche di sistemi concorrenti e distribuiti facciamo un passo indietro, recuperando uno strumento di cui abbiamo già parlato, rileggendolo sotto un'altra luce.
L'astrazione di esecuzione fornita dallo Stream si basa sull'inversione del controllo dell'iterazione, dato che lo Stream avanza nell'esecuzione e decide la sua struttura. Manca però un protocollo esplicito di gestione della terminazione dello stream e gli errori sono gestiti come eccezioni.

Per andare oltre a queste situazioni, nel 2009 sono state inventate le *Reactive Extensions* (oppure *rxJava*). È più una sorta di API, definendo come il codice venga eseguito ed una semantica di definizione di elaborazioni asincrone di sequenze di oggetti, basandosi su un intero modello di esecuzione.
Il lavoro teorico ne spinge il porting verso altre piattaforme (es. Netflix che rilascia RxJava).



L'idea principale del modello *è l'Observer pattern* basato su un *Observer*, oggetto concettualmente simile ad uno Stream, che intende conoscere nel tempo lo stato (tramite notifica ad esempio) di un altro oggetto quando questo cambia. Un *Observable* poi è un oggetto che può prendere dati da una sorgente ed il cui stato può essere di interesse per altri oggetti. Essi possono essere bloccanti (*Blocking*) con chiamata sincrona, oppure non bloccanti (*Non-Blocking*) con esecuzione asincrona.

L'insieme dei metodi utili è:

evento	Iterable	Observable
successivo	`T next()`	`onNext(T)`
errore	lancia `Exception`	`onError(E)`
completamento	`!hasNext()`	`onCompleted()`

Prendiamo sempre il problema del filtraggio dei numeri perfetti, trattando lo stream come *Observable*:

```
System.out.println("Defining...");
Observable.range(0, 10000).map(new RxDivisors())
  .filter(new RxPerfectPredicate())
  .subscribe((c) -> {
    System.out.println(c);
  }, (t) -> {
    t.printStackTrace();
  }, () -> {
    System.out.println("Done");
  });
System.out.println("Defined");
```

A livello di esecuzione, è simile allo stream ma si ha:

1) una semantica più ricca
2) maggiore regolarità nella composizione (esecuzione/terminazione)
3) indipendenza dal modello di esecuzione

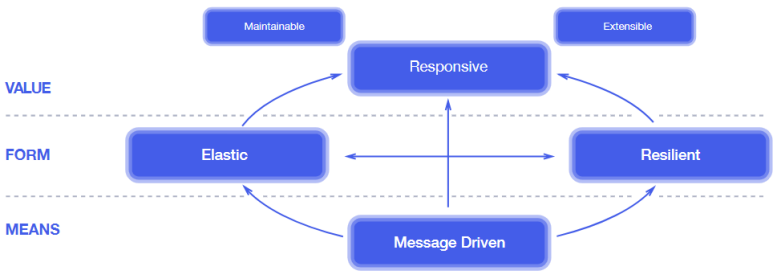
La maggior parte degli operatori sugli Observable accetta uno Scheduler come parametro ed ogni operatore può essere reso concorrente; in base allo Scheduler scelto si indica un certo tipo di concorrenza.

```
System.out.println("Defining...");
Observable.range(0, 1000000).map(new RxDivisors())
  .filter(new RxPerfectPredicate())
  .subscribeOn(Schedulers.computation())
  .subscribe(
    (c) -> { System.out.println(c); },
    (t) -> { t.printStackTrace(); },
    () -> { System.out.println("Done"); done[0] = true; });
System.out.println("Defined");
while (!done[0]) Thread.sleep(1000);
System.out.println("End");
```

Un Subscriber rappresenta un ascoltatore di un Observable, dato che fornisce il codice che reagisce agli eventi ed ottiene il risultato finale nella catena di elaborazione. Un Subject può consumare uno o più Observable, per poi comportarsi esso stesso da Observable e introdurre modifiche sostanziali nel flusso di eventi. Lo schema concettuale proposto da Rx è utile per:

1) costruire stream di elaborazione complessi ed asincroni
2) fornire un'interfaccia semplice in grado di trattare successioni di eventi
3) scrivere algoritmi facili tra un linguaggio e l'altro
4) strutturare un'elaborazione concorrente di uno stream di dati su garanzie solide
5) gestire gli eventi provenienti da una UI con la stessa semplicità di dati da file o altre sorgenti

L'insieme di principi su cui si basano le Reactive Extensions è il *Reactive Manifesto*.



Esso definisce come caratteristiche:

1) Pronti alla risposta (*Responsive*): Un sistema reattivo deve quindi dare sempre una risposta in tempo prevedibile, costante e soprattutto in tempi brevi, tale che sia le risposte che gli errori siano individuati e comunicati con le stesse tempistiche.
2) Resilienti (*Resilient*): Gestire i fallimenti continuando a rispondere con la stessa prontezza, dato che la resilienza si ottiene con la replicazione di componenti isolati, sapendo che il sistema può fallire e possono essere creati nuovi elementi atti a sostituire quelli errati.

Scritto da Gabriel

3) Elastici (*Elastic*): Consumare una quantità variabile di risorse in funzione del carico di ingresso, distribuendo il carico su più risorse possibile, senza colli di bottiglia o punti di conflitto, realizzando una scalabilità efficace ed economica grazie alla suddivisione dell'input su *shard.*
4) Orientati ai messaggi (*Message Driven*), perché questa primitiva abilita le altre caratteristiche. Attraverso lo scambio di messaggi i componenti posso rimanere disaccoppiati, indirizzando messaggi anche su nodi distribuiti, con un carico suddiviso tra copie di esecuzione su nodi differenti, consumando al meglio le risorse disponibili (tramite questo il sistema si struttura in forma Elastica e Resiliente per ottenere il valore della Prontezza alla risposta)

Oggi il problema del *Big Data* è molto grande e non può più essere adottata nello stesso modo. L'idea di esecuzione era data dai file *batch* (prendendo un insieme di dati e trasformandoli mandando poi un risultato) massivamente paralleli; tuttavia esso non funziona in quanto la latenza potrebbe essere superiore al periodo di utilità delle informazioni estratte.

In letteratura si comincia a parlare di *Fast Data*: dati di dimensione paragonabile al Big Data in arrivo continuo; è impensabile/inutile persisterli ed elaborarli (perché ormai obsoleti). Per il loro volume, velocità di arrivo e bassa latenza richiesta nel reagire alle informazioni estratte, è necessario trattarli in diretta a mano a mano che si presentano. Il componente più lento diventerà collo di bottiglia, stabilendo la velocità massima dell'elaborazione, per poi fallire data la velocità di arrivo dei dati.

Da questo problema, nasce la necessità di aggiungere alla semantica delle Reactive Extensions il concetto della *back-pressure*, quindi il controllo del flusso e la resistenza che il componente successivo può opporre ai dati provenienti dal componente precedente, permettendo ad ogni nodo di dichiarare quanti dati gestire.

L'obiettivo dei *Reactive Streams* è di governare lo stream di dati asincrono assicurandosi che chi riceve i dati non debba accumularne quantità illimitate da elaborare; tramite la back-pressure, i componenti non si devono trovare per forza su uno stesso nodo. Si parte da un modello semantico e da un kit di compatibilità di verifica dell'implementazione *(TCK, Technology Compatibility Kit*), con apposite interfacce applicative e protocolli di rete: la back-pressure è necessaria tra thread e nodi di calcolo distribuito.

Idee che implementano il TCK (reactive streams): *MongoDB Java Driver, RxJava, Java9 java.util.concurrent.flow.*

Il modello concettuale si basa su alcune interfacce utili:

1) *Publisher*: Fornisce un numero potenzialmente infinito di elementi in sequenza, rispettando le sue richieste

```
public interface Publisher< T > {
  public void subscribe(Subscriber< ? super T > s);
}
```

2) *Subscriber*: Consuma gli elementi forniti da un Publisher controllando il flusso degli elementi in arrivo

```
public interface Subscriber< T > {
  public void onSubscribe(Subscription s);
  public void onNext(T t);
  public void onError(Throwable t);
  public void onComplete();
}
```

3) *Subscription*: Controllo del legame tra Subscriber ed un Publisher, ad esempio interrompendolo

```
public interface Subscription {
  public void request(long n);
  public void cancel();
}
```

4) *Processor*: Esso è sia un Subscriber che un Publisher, sottostando ad entrambi i contratti. Esso è uno snodo di elaborazione intermedio in grado di alterare il flusso di elementi, aggregando più Publisher o controllando più Subscriber

```
public interface Processor< T, R >
  extends Subscriber< T >, Publisher< R > {
}
```

L'API dei reactive Streams è stata inclusa, partendo da Java 9 usando come base *java.util.concurrent.Flow*. Vediamo alcuni esempi, che fanno riferimento alla documentazione delle Reactive Extensions (a seconda della classe che si osserva si ha un certo comportamento).

1) *Map*: trasforma gli elementi di uno stream, ottenendo uno stream di elementi trasformati
2) *Flatmap*: trasforma gli elementi di uno stream, concatenando i risultati in un solo stream
3) *Filter*: emette uno stream contenente solo gli elementi che soddisfano un predicato
4) *Skip*: emette uno stream e salta i primi N elementi della sorgente
5) *Zip*: emette uno stream che combina coppie di elementi di due stream in ingresso
6) *Debounce*: emette un elemento solo se è passato un lasso di tempo dall'ultimo elemento della sorgente
7) *Window*: emette uno stream di partizioni dello stream sorgente

Metodo	Schedulatore
.io()	Per stream legati alle operazioni di IO
.single()	Usa un singolo thread
.computation()	Per operatori legati al calcolo
.from(ex)	Usa l'`Executor` fornito

L'asincronia nell'esecuzione dei vari operatori è data dallo Schedulatore usato per osservare un *Observable*, definendo un operatore di uno stream (ogni implementazione fornisce degli schedulatori in relazione alla piattaforma su cui opera, ad esempio RxJava che li ottiene tramite metodi statici dell'oggetto *Schedulers*).

In RxJava, l'operatore *parallel()* indica che uno stream da un certo punto in poi si costruisce in parallelo, avendo solo alcuni operatori consentiti (per questioni semantiche si fa questo ed occorre specificare che schedulatore si usa con *.runOn(scheduler)* o il metodo *sequential()*, intendendo un pezzo come sequenziale. Dunque, alcune sezioni delle pipeline possono essere configurate parallelamente. Il risultato è:

```
System.out.println("Defining...");
Flowable.range(0, 1000000).parallel(4)
  .runOn(Schedulers.computation()).map(new RxDivisors())
  .filter(new RxPerfectPredicate())
  .sequential().subscribe((c) -> {
      System.out.println(c);
    }, (t) -> {
      t.printStackTrace();
    }, () -> {
      System.out.println("Done");
      done[0] = true;
    });
System.out.println("Defined");
while (!done[0]) Thread.sleep(1000);
```

Attori

L'implementazione da noi esaminata non è conforme al Reactive Manifesto perché non basata sullo scambio di messaggi. Questo modello di esecuzione era peculiare e nato per requisiti ancora più specifici per soddisfare l'esecuzione Reactive.
Esaminiamo dunque un altro paradigma; il *modello ad attori*, creato nel 1963 in relazione ai primi modelli orientati agli oggetti (si richiama il linguaggio *Erlang* e la sua piattaforma di runtime OTP, creato da Ericsson, con un'affidabilità 9/9 (31 msec di guasto all'anno). Nonostante il lessico sia differente, il modello risultante è praticamente identico e di riferimento per implementazione negli altri linguaggi.

Un *Attore* è un'unità indipendente di elaborazione dotata di stato privato; comunica unicamente con messaggi diretti ad altri attori di cui si conosce il nome. In ogni momento ha un Comportamento che definisce la sua reazione ad un determinato messaggio. Generalmente può:

1) mutare il proprio stato interno
2) creare nuovi attori
3) inviare messaggi ad attori noti (numero di messaggi/numero di attori finito)
4) cambiare il suo comportamento

All'interno dell'attore non si ha concorrenza, dato che l'esecuzione è strettamente sequenziale.
È preferibile che la risposta sia il più veloce possibile.

Al di fuori dell'attore tutto è concorrente e distribuito, in quanto un attore può trovarsi dovunque ed è concorrente a tutti gli altri, così come i messaggi. Il fallimento di un attore è un'eventualità assolutamente normale, senza conseguenza sulla stabilità del programma (se fallimento stabile anche il modello di riferimento è stabile). Normalmente un Attore può (non deve per forza) supervisionare gli attori che ha creato, venendo loro notificato il fallimento (la notifica è un messaggio per cui un supervisore predispone un certo comportamento).

Le conseguenze di questo approccio sono diverse portando un sistema in cui:

1) l'avvio di nuovi processi/attori è estremamente economico
2) l'affidabilità è molto elevata
3) le performance e l'efficienza sono molto alte
4) la scalabilità è lineare o quasi (posso aspettarmi che affiancando un'altra macchina si possa ottenere il doppio della quantità di lavoro fatta per unità di tempo)
5) la distribuzione e la concorrenza sono caratteristiche primarie

In questo modo grandi moli di dati che arrivano in modo asincrono sono modellati come eventi, abilitando risultati difficili da raggiungere in altro modo.

Le caratteristiche degli Attori hanno un costo di natura concettuale; modellare un algoritmo come l'interazione di un insieme di attori concorrenti non è sempre percorribile come approccio, ma si ha un risultato efficiente (facile ed economico il fallimento, ma il lavoro per ripararlo può avere complessità maggiore). La forma mentis di risoluzione dei problemi, se condotta in modo tradizionale, può rivelarsi inutile e dannosa, dato che l'interazione non può fare nessuna ipotesi sull'ordine di ricezione dei messaggi o della loro affidabilità; ciò costringe a dover considerare attentamente il modello del risultato e le varie possibilità di fallimento.

Per iniziare con alcuni esempi è stato descritto da Edoardo Vacchi nel 2021; segue il link di riferimento:
https://evacchi.github.io/java/records/jbang/2021/11/16/write-you-a-chat-java-17-actor.html
https://www.javaadvent.com/2021/12/type-you-an-actor-runtime-for-greater-good-with-java-17-records-switch-expressions-and-jbang.html

1) Indirizzo di un attore

```
public interface Address < T > {
  Address < T > tell(T msg);
}
```

Questa interfaccia rappresenta un riferimento ad un attore. La parametrizzazione è sul tipo di messaggio che può essere inviato.

2) Comportamento (behavior) di un attore

```
public interface Behavior < T >
  extends Function < T, Effect < T > > {}
```

Il comportamento di un attore è restituire, alla ricezione di un messaggio, la funzione che produrrà il prossimo comportamento partendo dall'attuale.

Scritto da Gabriel

3) Effetto della ricezione di un messaggio

```
public interface Effect < T >
  extends Function<Behavior < T >, Behavior < T > >
```

Si ha un cambiamento di comportamento; quindi, è una funzione dal comportamento attuale a quello successivo. In questo modo può essere gestita in questo punto anche la modifica dello stato, intesa come trasformazione dello stato precedente in uno stato nuovo.

4) Effetto: mantenimento del comportamento attuale

```
static < T > Effect < T > stay() {
  return old -> old;
}
```

Ritorna una funzione identità, che ritorna esattamente il comportamento precedente.

5) Effetto: sostituzione del comportamento attuale

```
static < T > Effect < T > become(Behavior < T > next) {
  return current -> next;
}
```

Viene ritornato il comportamento impostato, sostituendo in tal modo il precedente.

6) Effetto: l'attore diventa inattivo

```
static < T > Effect < T > die() {
  return become(
    msg -> {
        return stay();
    });
}
```

Questo comportamento consuma il messaggio ignorandolo e rimane costante. L'attore non risponde più ad alcun messaggio.

7) Sistema: produce attori

```
public record System(Executor executor) {

  public < T > Address < T > actorOf(
    Function< Address < T >, Behavior < T > > initial) {
```

Il sistema richiede un Executor per gestire l'esecuzione delle azioni da parte degli attori. Il suo metodo principale è la creazione di un nuovo attore.

8) Costruzione dell'attore: gestione della concorrenza

```
abstract class AtomicRunnableAddress< T >
  implements Address < T >, Runnable {
    AtomicInteger on = new AtomicInteger(0);
}
```

Se *on* vale 1, l'attore sta già elaborando un messaggio.

9) Costruzione dell'attore: inizializzazione

```
var addr = new AtomicRunnableAddress< T >() {
  final ConcurrentLinkedQueue< T > mbox =
    new ConcurrentLinkedQueue<>();
  Behavior < T > behavior= initial.apply(this);
```

L'attore viene costruito come classe anonima. *mbox* è la coda dei messaggi in arrivo, *behavior* il comportamento corrente che viene inizializzato.

10) Costruzione dell'attore: ricezione di un messaggio

```
public Address<T> tell(T msg) {
  mbox.offer(msg);
  async();
  return this;
}
```

Ritorna *this* per permettere la concatenazione di più messaggi.

11) Costruzione dell'attore: elaborazione di un messaggio

```
void async() {
  if (!mbox.isEmpty() && on.compareAndSet(0, 1)) {
    try {
      executor.execute(this);
    } catch (Throwable t) {
      on.set(0);
      throw t;
    }
  }
}
```

Se c'è un messaggio da consumare, e non ne stiamo già elaborando uno, accodiamoci per l'esecuzione.
L'Executor è accessibile perché questa è una closure sul record System.

12) Costruzione dell'attore: esecuzione di un messaggio

```
public void run() {
  try {
    if (on.get() == 1)
      behavior = behavior.apply(mbox.poll()).apply(behavior);
  } finally {
    // processing complete, the actor is inactive
    on.set(0);
    async();
  }
}
```

Se non ci sono altre elaborazioni in corso, behavior.apply() ci ritorna l'effetto che, applicato a behavior, ritorna a sua volta il prossimo comportamento. Completiamo l'elaborazione, e verifichiamo la disponibilità di altri messaggi.
Schema fatto dal prof:

B : M → E (ottengo l'effetto della funzione del comportamento corrente)
B → B (ottengo l'effetto del comportamento nuovo)

13) Costruzione dell'attore: conclusione

```
  return addr;
}
```

L'attore è completo, e viene restituito come indirizzo richiamabile.

L'oggetto System ci permette quindi di costruire degli attori a cui possiamo associare un comportamento iniziale ed inviare messaggi.

Esempio 1: accetta un messaggio e poi si spegne

```
Address < String > actor =
  actorSystem.actorOf(
    self -> msg -> {
      out.println("got msg: '" + msg);
      return Effect.die();
    });

actor.tell("foo").tell("bar");
```

Notare la costruzione "curried" della funzione dello stato iniziale, cioè la tecnica di conversione di una funzione che accetta più argomenti in una sequenza di funzioni in cui ciascuna accetta un singolo argomento.

Esempio 2: Ping-pong

```
record Ping(Address < Pong > sender) {};

sealed interface Pong {};

record SimplePong(Address < Ping > sender) implements Pong {};
record DeadlyPong(Address < Ping > sender) implements Pong {};
```

Il compilatore sa che le due implementazioni proposte sono le uniche possibili per l'interfaccia Pong.

```
var actorSystem = new System(Executors.newCachedThreadPool());
Address < Ping > ponger = actorSystem.actorOf(
  self -> msg -> pongerBehavior(self, msg, 0));
Address < Pong > pinger = actorSystem.actorOf(
  self -> msg -> pingerBehavior(self, msg));
ponger.tell(new Ping(pinger));
```

Costruiamo due attori: uno riceve messaggi di tipo Ping, l'altro di tipo Pong. Ogni messaggio contiene l'indirizzo del mittente.

```
static Effect< Ping > pongerBehavior(
  Address< Ping > self, Ping msg, int counter) {
  return switch (msg) {
    case Ping p && counter < 10 -> {
      p.sender().tell(new SimplePong(self));
      yield Effect.become(m ->
        pongerBehavior(self, m, counter + 1));
    }
    case Ping p -> {
      p.sender().tell(new DeadlyPong(self));
      yield Effect.die();
    }
  };
}
```

L'attore Pong ha un contatore: se ha ricevuto meno di dieci Ping, ritorna una risposta normale. Altrimenti ritorna una Poison Pill per segnalare all'altro attore di terminare. Notate come il contatore sia gestito unicamente come valore della closure, e non come variabile dello stato.

L'attore Pinger risponde ai due possibili messaggi. Non è necessario un caso di default perché l'interfaccia Pong è sealed; quindi, il compilatore può dimostrare l'esaustività dello switch.

```
static Effect<Pong> pingerBehavior(Address<Pong> self, Pong msg) {
  return switch (msg) {
    case SimplePong p -> {
      p.sender().tell(new Ping(self));
      yield Effect.stay();
    }
    case DeadlyPong p -> {
      p.sender().tell(new Ping(self));
      yield Effect.die();
    }
  };
}
```

L'uso della closure come contenitore di stato deriva dal dualismo closure/oggetto, cioè tramite funzioni anonime incapsuliamo lo stato corrente degli oggetti, ma similmente gli oggetti incorporano già tutti gli aspetti della closure (che sappiamo essere una sorta di raccoglitore di questi); poi tutto dipende dall'implementazione fatta dopo.

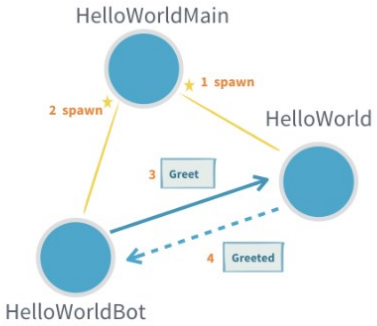
Concretamente il lessico degli attori si riferisce al framework *Akka*, molto più complesso da costruire ed organizzare. Esso fu inventato nel 2009, prendendo come modello esplicito per implementare sulle JVM un modello ad attori supportando libreria scalabile.

Il framework ha varie caratteristiche:

1) compatibile con l'ecosistema della JVM
2) scritto in Scala ma usabile anche con una API in Java
3) il modello di supervisione degli attori è obbligatoriamente da parte del "genitore", cioè solo l'attore che ne ha creato un altro può supervisionarlo.

Gli attori comunicano fra loro in modo pilotato dai tipi: ciò codifica l'interazione nelle caratteristiche delle classi usate per guidare l'interazione ed è il compilatore stesso ad impedirci di inviare messaggi che non possono essere ricevuti.
Akka supporta un'implementazione di Reactive Streams che sfrutta la scalabilità del sistema per raggiungere prestazioni considerevoli sfruttando un modello più semplice degli attori.

Vediamo un esempio completo per capire le complessità del modello:



In questo sistema, un attore manda un saluto, e un altro risponde. Il primo attore ripete per un numero massimo di volte. Un attore supervisore avvia il colloquio.

Come prima cosa dobbiamo definire i messaggi di saluto:

```
public static final class Greet {
  public final String whom;
  public final ActorRef< Greeted > replyTo;
  public Greet(String whom, ActorRef< Greeted > replyTo) {
    this.whom = whom;
    this.replyTo = replyTo;
  }
}
```

Questo è il messaggio di saluto. replyTo è il mittente.

e di risposta:

```
public static final class Greeted {
  public final String whom;
  public final ActorRef< Greet > from;

  public Greeted(String whom, ActorRef< Greet > from) {
    this.whom = whom;
    this.from = from;
  }
}
```

Una ActorRef è un riferimento ad un attore, che accetta quel tipo di messaggio.

Dobbiamo poi definire come si comporta l'attore che saluta al momento della creazione:

```
public class HelloWorld extends
  AbstractBehavior< HelloWorld.Greet > {

  public static Behavior< Greet > create() {
    return Behaviors.setup(HelloWorld::new);
  }

  private HelloWorld(ActorContext< Greet > context) {
    super(context);
  }
```

Behavior è la classe base dei comportamenti degli attori. Indichiamo che l'attore, alla creazione, usa il suo costruttore.

Dobbiamo inoltre definire come si comporta al ricevimento di un messaggio:

```
@Override
public Receive< Greet > createReceive() {
  return newReceiveBuilder().onMessage(Greet.class,
    this::onGreet).build();
}

private Behavior< Greet > onGreet(Greet command) {
  getContext().getLog().info("Hello {}!", command.whom);
  command.replyTo.tell(new Greeted(command.whom,
    getContext().getSelf()));
  return this;
}
```

createReceive() è un metodo che fornisce il comportamento. Alla ricezione di un messaggio di tipo Greet, il comportamento è il metodo onGreet(command). Il quale individua il mittente, e gli risponde con un altro messaggio. Il compilatore ha le informazioni per impedirci di inviare un messaggio di tipo sbagliato.

Ora si mostra l'attore che invia il saluto. Il comportamento alla creazione viene definito (questo attore ha uno stato privato, e la sua creazione richiede un parametro. Dato che lo stato è una variabile semplice, non abbiamo problemi di concorrenza).

```
public class HelloWorldBot
  extends AbstractBehavior< HelloWorld.Greeted > {

  public static Behavior< HelloWorld.Greeted >
    create(int max) {
    return Behaviors.setup(c -> new HelloWorldBot(c, max));
  }
  private final int max;
  private int greetingCounter;

  private HelloWorldBot(ActorContext< HelloWorld.Greeted >
    ctx, int max) {
    super(context); this.max = max;
  }
```

```
@Override
public Receive< HelloWorld.Greeted > createReceive() {
  return newReceiveBuilder().onMessage(
    HelloWorld.Greeted.class, this::onGreeted).build();
}
private Behavior< HelloWorld.Greeted >
  onGreeted(HelloWorld.Greeted message) {
    greetingCounter++;
    getContext().getLog().info("Greeting {} for {}",
      greetingCounter, message.whom);
    if (greetingCounter == max) {
      return Behaviors.stopped();
    } else {
      message.from.tell(
        new HelloWorld.Greet(message.whom,
          getContext().getSelf()));
    return this;
  }
}
```

Se abbiamo raggiunto il massimo, interrompiamo le risposte terminando l'esecuzione dell'attore. Altrimenti rispondiamo a chi ci ha inviato il saluto, impostando il messaggio di saluto e recuperando il contesto dell'invocazione della chiamata.

L'attore che coreografa l'interazione deve rispondere ad un messaggio diverso, salutando sulla base del nome passato come parametro.

```
public class HelloWorldMain
  extends AbstractBehavior< HelloWorldMain.SayHello> {

  public static class SayHello {
    public final String name;

    public SayHello(String name) {
      this.name = name;
    }
  }
```

```
private final ActorRef< HelloWorld.Greet > greeter;

public static Behavior< SayHello > create() {
  return Behaviors.setup(HelloWorldMain::new);
}

private HelloWorldMain(ActorContext< SayHello > context) {
  super(context);
  greeter = context.spawn(HelloWorld.create(), "greeter");
}
```

Il suo stato è il riferimento all'attore creato al momento della costruzione, creando il contesto del saluto.

```
@Override
public Receive< SayHello > createReceive() {
  return newReceiveBuilder().onMessage(SayHello.class,
    this::onSayHello).build();
}

private Behavior< SayHello > onSayHello(SayHello command) {
  ActorRef< HelloWorld.Greeted > replyTo = getContext()
    .spawn(HelloWorldBot.create(3), command.name);
  greeter.tell(new HelloWorld.Greet(command.name, replyTo));
  return this;
}
```

Al ricevimento del messaggio d'avvio, crea l'attore di risposta ed invia il primo saluto per avviare la conversazione.

La classe principale costruisce il sistema di attori ed avvia il coreografo, leggendo lo stato degli attori ed eseguendo le operazioni utili.

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

  final ActorSystem< HelloWorldMain.SayHello > greeterMain =
    ActorSystem.create(HelloWorldMain.create(), "helloakka");
  greeterMain.tell(new HelloWorldMain.SayHello("Charles"));

  try {
    System.in.read();
  } catch (IOException ignored) {
  } finally {
      greeterMain.terminate();
  }
}
```

Lezione 22: Esecuzione alternativa

Abbiamo analizzato quattro diversi paradigmi, ovvero quattro diversi modi di organizzare il codice per ottenere diverse caratteristiche funzionali nella risoluzione di un problema.

1) La concorrenza ci permette di sfruttare meglio le risorse.
2) La distribuzione ci permette di espanderci oltre il singolo nodo di calcolo.
3) La reattività ci permette di minimizzare la latenza di risposta.
4) Il modello ad attori ci permette di astrarre oltre l'asincronia e la distribuzione.

In tutti questi ambienti, abbiamo usato l'ambiente di esecuzione consueto di Java, compilando in bytecode ed eseguendolo con la JVM, che applica le strategie JIT (Just In Time, quindi la JVM richiama direttamente il codice compilato del metodo, permettendo di essere eseguito ovunque) per bilanciare l'efficienza di esecuzione; tuttavia questo ha un costo in termini di avvio del programma.

La JVM può aver bisogno anche di secondi prima di avere a disposizione dati sufficienti per cominciare ad effettuare delle scelte relative a quale codice trasformare con JIT. Inoltre, Il codice iniziale è necessariamente interpretato, e quindi più lento. E anche la JIT ha il suo costo in termini di prestazioni. Oggi tutto ciò non è più accettabile in termini di performance.

I primi siti web dinamici erano composti da una richiesta di I/O da parte di un browser, solitamente composto da uno script. Il programma viene eseguito, prendendo la richiesta dello standard input da un programma e dando come output la risposta al browser.

Tale modalità di funzionamento è formalizzata come *Common Gateway Interface* (tuttora in vigore per vari linguaggi, es. Perl e Python). Molto presto diventa evidente la penalità di performance dovuta all'avvio di un processo esterno al web server. Nasce *l'application server*, per mantenere sempre pronte alla risposta le applicazioni (spesso legato ad una certa tecnologia, dunque spesso nascono problemi di compatibilità). Java e la JVM quindi nascono in una situazione in cui è normale per un server rimanere in esecuzione, in attesa di richieste. L'efficienza si persegue gestendo più applicazioni sulla stessa macchina, che ne condividono le risorse.

Oggi un servizio cloud cosiddetto *serverless* acquisisce le risorse necessarie all'esecuzione di una sola richiesta, pagando i millisecondi necessari alla risposta del codice. Il tempo di startup e la dimensione di un application server, modalità tipiche di erogazione di molti tipi di applicazioni, diventano uno svantaggio. Un server non è più sempre in attesa delle richieste: esiste per il tempo strettamente necessario ad erogare una sola risposta. Per la JVM è un problema particolarmente pressante: sebbene le prestazioni a regime siano di ottimo livello, lo diventano solo dopo diversi secondi di attività.

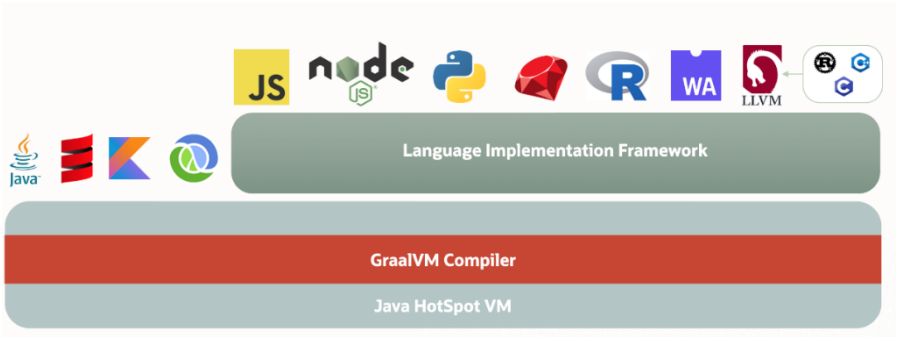
La risposta a questo problema è il progetto *GraalVM*, che nasce come progetto di riscrittura del compilatore Java in Java stesso (perché non era *bootstrapped*, non scritto nello stesso linguaggio del codice). Si focalizzava inizialmente sulla sola parte di frontend (parte grafica) del compilatore.

Scritto da Gabriel

In seguito a questo cambio di rotta, diventa un sistema completo che include:

1) un nuovo compilatore JIT (che può sostituire quello ufficiale di Java)
2) un compilatore ahead-of-time per Java (per ridurre il tempo effettivo di run time del codice e compiendo a tutti gli effetti una precompilazione)
3) una specifica di codice intermedio ("Truffle"), che trasforma il linguaggio in una rappresentazione indirizzata ad un backend che fa l'ultimo passaggio in codice nativo
4) un back-end che usa il sistema LLVM/Low Level Virtual Machine (infrastruttura di compilazione con cui il programmatore può creare una macchina virtuale, un compilatore per una architettura specifica e software di ottimizzazione del codice indipendenti dal tipo di linguaggio utilizzato o dalla piattaforma) ed un runtime minimale per ottenere un eseguibile nativo da una rappresentazione Truffle
5) strumentazione che non deve conoscere dettagli in merito al debug e all'ottimizzazione

L'idea grafica è:



GraalVM è in grado di compilare un qualsiasi bytecode da un linguaggio sulla JVM in un eseguibile nativo. Attraverso Truffle, può supportare anche altri linguaggi (tramite front-end appositi).
La compilazione ahead-of-time, in un linguaggio come Java, ha vari problemi: in particolare, il sistema di Classloading e le feature di *reflection* (insieme di API che ispezionano la struttura delle classi, con varie librerie e strumenti che richiedono adattamenti utili per funzionare) devono essere limitate per rendere prevedibile il codice disponibile al runtime.

Quello che si ottiene però è una sostenuta riduzione della dimensione dell'eseguibile finale ma soprattutto del tempo di avvio (performance che potrebbero essere a rigore leggermente inferiori non avendo adeguate informazioni a runtime rispetto a JIT; per esempio anche in merito alla garbage collection, sapendo che il programma termina comunque).

Attraverso un framework che supporta GraalVM, siamo in grado di adattare un precedente esempio per osservarne un eseguibile nativo già impacchettato in un container.
Scegliamo per questo esempio il framework *Micronaut,* con supporto nativo al cloud particolarmente *opinionated* (fa scelte precise ed indirizza correttamente all'uso degli strumenti giusti) e propone una impostazione precisa di progetto, scegliendo all'interno dell'insieme delle tecnologie supportate le implementazioni delle funzioni necessarie ed integrando le componenti compatibili con il processo *ahead-of-time*.

Questo approccio è dettato dalla necessità di selezione ed integrazioni di componenti utilizzabili con il processo ahead-of-time, preparando in maniera personalizzata un progetto, come si vede da:
https://micronaut.io/launch/
La parte più interessante è l'interfacciamento con il framework, in particolare la dichiarazione delle rotte. Micronaut ha in questo un approccio molto differente da VertX; in modo decisamente più tradizionale, usa delle annotazioni (o note al momento della compilazione, ed esiste già un supporto nel compilatore per analizzarle come passaggio di compilazione per produrre metadati che possono essere usati dai passi successivi). Usiamo come riferimento: https://hg.sr.ht/~michelemauro/pdp2021-mnaut

Scritto da Gabriel

La classe server è annotata con *@Controller* per indicare che contiene metodi che serviranno richieste Web. Le costanti sono le stesse dell'altro esempio evergreen TicTacToe.

```
@Controller("/")
public class Server {

  private static final String GAME_NOT_FOUND = ...
  private static final String JOIN_FORM = ...
  private static final String WAIT_FOR_ANOTHER = ...

  static final ObjectMapper mapper = new ObjectMapper();
@Get(value = "/", produces = MediaType.TEXT_HTML)
public String welcome() {
   return JOIN_FORM;
}
```

Tutte le classi annotate con *@Controller* vengono raccolte all'avvio del server e la struttura delle API esposte viene costruita analizzando il loro contenuto. Questo lavoro viene fatto al runtime se il framework viene avviato normalmente (cioè come applicazione sulla JVM) o al momento della compilazione se viene generato un eseguibile nativo.

```
@Post(value="/game",
  consumes = MediaType.APPLICATION_FORM_URLENCODED,
  produces=MediaType.TEXT_HTML)
public HttpResponse<String> join() {
    GameLocation location =
      new GameLocation(gameServer.create());
    return HttpResponse.redirect(
      URI.create(location.game));
}
```

Le annotazioni indicano i metodi che devono servire le varie richieste, specificando il pattern di URL a cui rispondono, i formati trattati e l'interfacciamento con la richiesta.
La risposta invece è strutturata con una *join*, che simula le variabili utili all'esecuzione e crea una nuova locazione di gioco, reindirizzando la risposta con un URI corretto.

Il pattern può contenere variabili, che possono essere trasformate direttamente in parametri del metodo. L'annotazione ha accesso al nome lessicale del parametro, e quindi può riconoscerlo dentro il pattern. La risposta è costruita a partire dal valore di ritorno del metodo; se complessa deve essere contenuta in un oggetto adeguatamente espressivo.

```
@Get(value = "/game/{playerId}",
  produces = MediaType.TEXT_HTML)
public HttpResponse< String > gameStatus(
  @PathVariable String playerId) {

  boolean open = gameServer.open(playerId);
  if (!open)
    return HttpResponse.notFound().body(GAME_NOT_FOUND);

  String result = gameServer.status(playerId)
    .map((res) -> render(playerId, res.idx, res.status))
    .orElse(String.format(WAIT_FOR_ANOTHER, playerId));
  return HttpResponse.ok().body(result);
}
```

I parametri vengono convertiti secondo una serie di convenzioni.
É possibile fornire al framework delle strategie specifiche per personalizzare la conversione.

Successivamente, recuperiamo lo stato, formattando correttamente il risultato della partita sulla base di un altro giocatore qualsiasi:

```
@Post(
  value = "/game{/playerId}",
  produces = MediaType.TEXT_HTML,
  consumes = MediaType.APPLICATION_FORM_URLENCODED)
public HttpResponse< String > game(
  @PathVariable @Nullable String playerId,
  @Nullable Integer move) {

  if (playerId == null) {
    GameLocation location = ...;
    return HttpResponse.status(HttpStatus.TEMPORARY_REDIRECT)
      .header("Location", location.game)
      .body("");
  }
```

Il metodo selezionato deve avere un pattern corrispondente all'url della richiesta, accettare un media type compatibile e produrne uno conforme alle richieste del client.
Recuperiamo la partita e relativo risultato sulla base del giocatore, attivando un redirect:

In questo caso la situazione è più complessa: il linguaggio del pattern non può esprimere come rotte differenti quella con e quella senza il parametro *playerId*; quindi, dobbiamo gestire nel metodo la sua presenza. Nel caso in cui il parametro playerId manchi, stiamo rispondendo ad una richiesta POST /game. Se invece è presente, stiamo rispondendo ad una richiesta *POST /game/123456*.

Scritto da Gabriel

Il metodo quindi deve essere compatibile con le richieste specifiche.

Le risorse sono gestite quindi in modo diverso da VertX; quest'ultimo va verso un modello reattivo, mentre *Micronaut* va verso un modello molto sincrono. Di fatto si va verso un approccio orientato ai processori ARM, processori di tipo RISC usati spesso nel mondo di telefoni, interessanti per come si adattano all'esecuzione. Il bytecode si genera verso set di istruzioni differenti e l'interazione fra linguaggio compilato nativo, i container che astraggono l'hardware, ora molto più legato all'I/O sta facendo abbassare sempre più il costo nella generazione dei set di istruzioni.

Esistono altri approcci per perseguire obiettivi simili a quelli di GraalVM, seguendo però strade differenti. Ne citiamo espressamente due:

1) *TornadoVM* è un plugin per varie implementazioni di JVM che permette di scrivere codice Java in grado di essere eseguito su GPU, FPGA ed altri hardware eterogenei, accedendo a paradigmi di calcolo di solito dominio esclusivo di linguaggi di basso livello o molto specializzati. La struttura può essere configurata sul campo in merito ai circuiti, specializzandoli per alcuni compiti ed eseguendoli.
2) *KotlinNative* si appoggia ad LLVM per produrre un eseguibile direttamente dal codice Kotlin, senza passare per la JVM .La struttura del linguaggio e del suo compilator e permettono questo passaggio. Il risultato è un eseguibile nativo, compilato ahead-of-time time con un approccio però diverso da GraalVM.

Lezione 23: Java 19

La release di Java 19 è prevista per settembre 2022 ed è sviluppato dal JCP (Java Community Process), articolato da JSR (Java Specification Requests) con uno specifico percorso di approvazione delle richieste. Dal 2011 nasce il JDK Enhancement Proposal per lanciare feature più velocemente (di solito ogni 6 mesi). Ora andiamo verso il JDK 2019 e nella nuova versione di Java individuiamo queste JEP:

JEP	Titolo
424	Foreign Function & Memory API (Preview)
426	Vector API (Fourth Incubator)
422	Linux/RISC-V Port

JEP	Titolo
405	Record Patterns (Preview)
427	Pattern Matching for switch (Third Preview)
425	Virtual Threads (Preview)
428	Structured Concurrency (Proposed)

Noi siamo interessati in particolare alle 425/428, 405/427 riguardano le evoluzioni sintattiche in merito ai Record. La *JEP 405 Record Patterns* propone una review per una sintassi per de-costruire i Record e usarne direttamente le proprietà; è un primo passo per implementare il Pattern Matching.

Prima della JEP 394:

```
record Point(int x, int y);

public void print(Object o) {
  if (o instanceof Point) {
    var ix = ((Point)o).x()
    var iy = ((Point)o).y()
    System.out.println("x,y: " + ix + "," + iy);
  }
}
```

Dopo la JEP 394:

```
record Point(int x, int y);

public void print(Object o) {
  if (o instanceof Point p) {
    System.out.println("x,y: " + p.x() + "," + p.y());
  }
}
```

Ulteriore esempio e sintassi possibile (dx):

```
enum Color {RED, GREEN, BLUE}
record ColoredPoint(Point p, Color color) {}
record Point(int x, int y) {}
record Square(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

```
public void printUpperLeftColoredPoint(Square s) {
  if (s instanceof Square(
    ColoredPoint(Point(var x, var y), var color), var leftRight
    ){
    // x, y, color, leftRight sono definiti
  }
}
```

L'obiettivo di questa JEP è di avvicinarsi alla sintassi equivalente a linguaggi funzionali, dato che il compilatore "capisce" dal contesto di invocazione in cui è chiamato il record, a che tipo convertire e a quale riferirsi.
Strettamente legato a questa, viene proposta una nuova sintassi per lo switch (JEP 427).
Confronto prima (sx) e dopo (dx) la JEP 427:

```
static void testTriangle(Shape s) {
  switch (s) {
      case Triangle t && t.calculateArea() > 100 ->
          System.out.println("Large triangle");
      case Triangle t ->
          System.out.println("Small triangle");
      default ->
          System.out.println("Non-triangle");
  }
}
```

```
static void testTriangle(Shape s) {
  switch (s) {
    case Triangle t when t.calculateArea() > 100 ->
      System.out.println("Large triangle");
    case Triangle t ->
      System.out.println("Small triangle");
    case null ->
      System.out.println("No shape");
    default ->
      System.out.println("Non-triangle");
  }
}
```

Viene introdotta la keyword *when*, tale da rendere più chiara la costruzione della semantica delle espressioni di guardia e dell'uso di *null*, capendo in maniera esaustiva il contesto delle classi.

La *JEP 425 Virtual threads* introduce uno degli obiettivi finali di Project Loom, le *fiber*.
Un Thread modella una linea di esecuzione di un processo nella JVM e abbiamo visto che, nella pratica, un Thread superiore a 10000 può mettere a dura prova la gestione delle risorse dell'OS, esaurendone alcune.
Per algoritmi con Blocking Factor vicino ad 1 (legati in prevalenza all'I/O), si cerca una soluzione che scriva codice per i Thread in relazione al tempo d'attesa possibile.
Un approccio è quello di riscrivere il codice in modo asincrono, in modo che le librerie gestiscano il passaggio delle operazioni dei Thread; si cerca quindi un'*unità di concorrenza* più piccola dei Thread.

Il risultato di questa ricerca sono i *virtual thread*, sintatticamente simili ai Thread normali, ma la libreria standard si comporta diversamente; la JVM gestisce la messa in attesa del virtual thread (*unmounting*), rimettendolo in esecuzione quando i dati sono disponibili (*mounting*).
In questo modo lanciamo contemporaneamente centinaia di migliaia di operazioni, senza preoccuparsi al limite di sistema.

```
try (var executor =
    Executors.newVirtualThreadPerTaskExecutor()) {
  IntStream.range(0, 10_000).forEach(i -> {
      executor.submit(() -> {
          Thread.sleep(Duration.ofSeconds(1));
          return i;
      });
  });
}
```

I virtual thread sono quindi ottimizzati per essere usati una volta sola, con le operazioni della libreria standard che hanno un comportamento bloccante che ne provoca lo smontaggio in maniera rapida e trasparente. Per esempio, nella successiva istruzione, il virtual thread può essere smontato tre volte:

```
response.send(future1.get() + future2.get());
```

Alcune operazioni non possono smontare dei virtual thread:

1) codice *synchronized*
2) codice *native* o interfacciato con una foreign function (cioè una funzione che non appartiene allo stesso linguaggio di programmazione su cui si opera)

La JEP si appoggia al lavoro già fatto per l'aggiornamento della libreria I/O, estendendolo alla libreria standard ed introducendo nuove classi. A questo scopo è stato introdotto il costrutto delle *coroutines*, cercando di mantenere un modello di programmazione che non divida metodi classici e metodi asincroni o per specifiche di limitazioni e limiti tecnici, mantenendo compatibilità nei vari ambienti. L'obiettivo della

JEP è di mantenere i vantaggi della concorrenza collaborativa, senza esporre all'utente l'esplicita gestione; tuttavia, in caso di compiti legati alla CPU, si può ottenere una *unfairness*, introducendo una sintassi esplicita di *yield*, indirizzata ai soli compiti di IO.

Altra JEP interessante è la JEP 428 Structured Concurrency, il cui scopo è strutturare la concorrenza indicando più compiti eseguiti da thread diversi fanno parte di un'unica unità di lavoro. Ad esempio:

```
Response handle() throws ExecutionException, InterruptedException {
  Future< String >  user  = es.submit(() -> findUser());
  Future< Integer > order = es.submit(() -> fetchOrder());
  String theUser  = user.get();   // Join findUser
  int    theOrder = order.get();  // Join fetchOrder
  return new Response(theUser, theOrder);
}
```

Gli scenari di fallimento sono:

- se findUser() fallisce, fetchOrder() viene perso
- se handle() viene interrotto, entrambi i Future vengono persi
- se findUser() è lento e fetchOrder() fallisce, si aspetta inutilmente su user.get()

Si vuole quindi cercare una struttura che descriva la relazione esistente tra i task e decidendo sulla base del loro specifico comportamento.

```
Response handle() throws ExecutionException, InterruptedException {
  try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Future< String >  user  = scope.fork(() -> findUser());
    Future< Integer > order = scope.fork(() -> fetchOrder());

    scope.join();           // Join both forks
    scope.throwIfFailed(); // ... and propagate errors

    // Here, both forks have succeeded, so compose their results
    return new Response(user.resultNow(), order.resultNow());
  }
}
```

La sintassi dei thread con *try-with-resources*, si introducono un'insieme di compiti unitari, in particolare:

- *short-circuiting* come gli stream
- propagazione delle cancellazioni
- chiarezza della sintassi
- osservabilità del risultato.

L'implementazione attuale risiede in un modulo separato del JDK e va esplicitamente importato per poterlo usare, con problematiche comunque legate ai compiti di IO.
Altre novità di Java 19:

1) *JEP 426: Vector API (Fourth Incubator)*, istruzioni specifiche per CPU *SIMD* (Single Instruction Multiple Data), studiando architetture che elaborano sequenze di istruzioni su insiemi di dati contemporaneamente
2) *JEP 427: Foreign Function e Memory API*, permettendo di sfruttare la memoria *off-heap* per migliorare l'interazione con codice non-Java tramite JNI (scomoda da usare)
3) *JEP 422: Linux/RISC-V Port* per set di istruzioni RISC-V

Lezione 24: Riepilogo

Java → Linguaggio orientato ad oggetti, eseguito su di una virtual machine in modalità mista: interpretato, e gradualmente compilato durante l'esecuzione (JIT)

JVM → La compilazione di un insieme di sorgenti Java è un processo deterministico, insensibile all'ordine con cui le classi si presentano al compilatore. Ogni classe deve apparire unicamente in una determinata posizione nel filesystem: quindi è semplice per il compilatore cercare una classe e decidere se l'ha trovata o meno. Il collegamento fra nomi e codice avviene dinamicamente al runtime: è considerato normale caricare nuovo codice durante l'esecuzione; in determinate condizioni è possibile sostituire, senza fermare il programma, codice esistente per caricarne una nuova definizione sotto lo stesso nome.

La sintassi di Java richiama i linguaggi della famiglia del C. Il codice è organizzato in classi, all'interno di metodi che sono identificati dal nome e dal numero e tipo degli argomenti.
I componenti di una classe Java possono avere una delle seguenti classi di visibilità:

1) *private*: solo altri componenti della stessa classe
2) *protected*: solo discendenti della classe
3) *default*: altre classi dello stesso package
4) *public*: tutte le altre classi

Gli errori applicativi sono gestiti tramite eccezioni, che sono oggetti non molto dissimili dagli altri.
È obbligatorio dichiarare la possibilità per un metodo di lanciare alcuni tipi di eccezioni. La coerenza di queste dichiarazioni è controllata dal compilatore.

Il *modello di ereditarietà* è asimmetrico: *singola* nei confronti delle classi, *multipla* nei confronti delle interfacce. Il compilatore considera un diamond problem un errore semantico bloccante.
Le classi definiscono un tipo di oggetto. Possono contenere variabili, metodi, inizializzatori e altre definizioni di classi. I componenti che sono unici per tutte le istanze di oggetti appartenenti alla classe sono definiti static.

Una *interfaccia* costituisce la dichiarazione di un contratto cui una classe può aderire. In questo senso, definisce un tipo. Una interfaccia può essere implementata da una classe anonima, specificando immediatamente il corpo dei metodi richiesti (quando una classe viene caricata, vengono eseguiti gli inizializzatori statici; quando un oggetto viene inizializzato, vengono eseguiti prima i costruttori della superclasse, poi gli inizializzatori di istanza e infine il resto del costruttore).
Una interfaccia può contenere metodi completi di implementazione, per i quali le classi aderenti non hanno l'obbligo di fornire un'implementazione. Una interfaccia con un solo metodo è detta *functional interface*.
Il compilatore permette di abbreviare la sintassi necessaria per istanziarla.

Una *annotazione* è una interfaccia speciale che può essere usata per aggiungere metadati a strutture sintattiche nel codice. Tali metadati possono essere usati durante la compilazione o al runtime, per cambiare il comportamento degli utilizzatori del codice stesso.
Il *record* è uno speciale tipo di oggetto immutabile, per cui il compilatore completa un insieme di metodi di default e non può essere usato per sincronizzazione.

Le classi possono accettare parametri di tipo detti *generici*. In questo modo è possibile scrivere codice largamente indipendente dallo specifico tipo usato durante l'esecuzione, facendo il minimo delle ipotesi possibile durante la scrittura e la compilazione. Non tutti i valori in Java sono oggetti: esistono alcuni tipi detti primitivi la cui rappresentazione è più semplice, e non costituisce un oggetto. Hanno di norma una versione cosiddetta *boxed*, che ha tutte le caratteristiche di un oggetto vero e proprio, ma è meno efficiente da utilizzare.

L'assenza di un valore oggetto è rappresentata con il valore *null*. Un valore primitivo non può essere assente, quindi nessuna variabile di tipo primitivo può assumere il valore *null*. Un tipo dotato di un numero limitato e definito a priori di valori è detto una *enumerazione*. Le enumerazioni permettono di modellare tipi di dati specifici e dotati di pochi valori costanti.
Le espressioni in Java producono un valore di un certo tipo, primitivo od oggetto. Il compilatore può in molti casi dedurre il tipo del risultato di una espressione e quindi richiedere meno precisione nella sua indicazione.

Le *istruzioni* in Java non producono valore. La loro esecuzione è garantita semanticamente, ma non sequenzialmente: il compilatore può permettersi grandi libertà nel riorganizzare il codice a scopo di efficienza, a meno di non richiedere diversamente con precise sintassi.

Una *lambda expression* è una sintassi che può sostituire, in certe situazioni, una interfaccia con un singolo metodo per ottenere una scrittura più compatta e leggibile. Il compilatore si occupa di individuare il tipo richiesto e completare sintetizzandola la dichiarazione.

Le istruzioni condizionali comprendono le classiche istruzioni *if* e *switch*, oltre ad una sintassi di switch che è considerata una espressione, e quindi produce un valore.
Le istruzioni di ciclo comprendono i classici *do/while, while, for*, oltre ad una sintassi di *for* in grado di riconoscere una collezione di oggetti ed attraversarla.

L'istruzione *try* è richiesta per circondare codice che lancia eccezioni specifiche e per dichiarare come gestirle. Una apposita sintassi permette di dichiarare istanze di oggetti che implementano *Closable*, per i quali verranno sintetizzate le corrette istruzioni di chiusura al termine del blocco indicato.

Paradigmi

Il corso ha analizzato i seguenti paradigmi di programmazione:

Risorse/Scopo	Diverso	Comune
Comuni	*Concorrenza*	*Parallelismo*
Isolate	*Rete*	*Distribuzione*

In aggiunta sono stati presentati i paradigmi:
1) Reattivo
2) Attori

Concorrenza: Gestione di risorse condivise fra linee di esecuzione in competizione fra loro
Parallelismo: Efficiente assegnazione di risorse condivise fra linee di esecuzione che si suddividono il carico di parti di un compito
In rete: Interazione fra nodi con ruoli differenti separati da una rete di comunicazione
Distribuzione: Interazione fra nodi appartenenti ad uno stesso sistema che collaborano per il raggiungimento di uno scopo
Reattivo: Semantica più ricca per l'elaborazione asincrona diffusi di dati non limitati
Attori: Semantica di alto livello per modellare sistemi distribuiti, aggressivamente concorrenti, orientati alla robustezza

Concorrenza

Il paradigma concorrente nasce per motivazioni economiche allo scopo di sfruttare in modo più completo tutte le parti dell'hardware di una stessa macchina, mitigando le diverse velocità relative di CPU, memoria e canali di comunicazione.
I sistemi concorrenti devono mitigare quattro problematiche principali:

1) Non determinismo
2) Starvation
3) Race conditions
4) Deadlock

Le *condizioni di Coffman* sono necessarie perché si produca un deadlock:

1) Mutual exclusion
2) Resource holding
3) No preemption
4) Circular wait

Gli *approcci alla concorrenza* si possono distinguere in:

1) Collaborativa
2) Pre-Emptive
3) Tempo reale
4) Orientata agli eventi

Come in molti altri sistemi, una linea di esecuzione è rappresentata in Java da un oggetto *Thread*. Un nuovo Thread può essere avviato per ottenere una linea di esecuzione indipendente e parallela al chiamante.
La JVM termina correttamente solo quando tutti i thread creati dall'utente terminano.

Un Thread attraversa un insieme di stati:

1) New
2) Runnable / Running
3) Blocked / Waiting / Timed Waiting
4) Terminated

L'interfaccia *Runnable* definisce un compito che può essere eseguito da una linea di esecuzione.
L'interfaccia *Callable* definisce un compito che produce un risultato.
Un *Executor* rappresenta una strategia di gestione di Threads a cui possono essere consegnati Runnables o Callables perché siano eseguiti. Richiedendo l'esecuzione di un Callable si ottiene un *Future*, cioè una rappresentazione del calcolo in corso, che può essere interrogato sul completamento e per ottenerne il risultato.
Per manipolare dati tramite più linee di esecuzione concorrenti è necessario usare tipi di dati specifici:

1) Variabili *Atomic*
2) Strutture dati concorrenti
3) Variabili *Thread Local*

La libreria standard propone una struttura dati di *Stream* che può distribuire il consumo e l'elaborazione degli elementi in modo parallelo, se necessario. La struttura può esaminare le caratteristiche di una catena di elaborazione e prendere decisioni su come eseguirla efficacemente.
L'interfaccia *Spliterator* può essere implementata per alimentare uno stream parallelizzabile.
L'interfaccia *Collector* invece è dedicata alle operazioni di riduzione di uno stream che, per efficienza, richiedono un accumulatore mutabile.

La valutazione del grado di parallelismo ottimale per una determinata pipeline di esecuzione deve tenere conto del *Blocking Factor* del compito da eseguire:

- uso costante della CPU: `BF = 0`
- blocco costante in attesa di I/O: `BF = 1`
- `#threads <= (#cores)/(1-BF)`

La sincronizzazione di più linee di esecuzione richiede l'uso di primitive apposite; in ordine di complessità ed espressività:

1) *synchronized*
2) *wait()/notify()*
3) Locks, Conditions
4) Semaphores

Deve essere realizzata con attenzione una possibile implementazione, in quanto dimostrarne la correttezza è molto difficile.

Distribuzione

Il paradigma distribuito nasce per ricercare:

1) scalabilità superiore a quella raggiungibile su di un singolo nodo
2) resistenza/ridondanza di fronte al guasto
3) localizzazione più vicina agli utenti

Le problematiche principali del paradigma sono:

1) concorrenza fra i nodi componenti
2) asincronia degli eventi
3) imperscrutabilità dei fallimenti

La comunicazione fra diversi nodi può essere modellata come uno scambio di messaggi. Vari metodi cercano, con alterni successi, di avvicinare questo modello alla chiamata di un metodo di un altro oggetto.

L'oggetto *Socket* modella il corrispondente concetto di connessione TCP/IP: un flusso di dati non limitato, bidirezionale, asincrono. I due lati della comunicazione devono concordare sul protocollo di scambio dati per gestire i turni di invio e lettura delle informazioni. La comunicazione trasporta esclusivamente bytes: quindi, è necessario aggiungere informazioni, o codificarle nel protocollo, per ricavare da questi oggetti usabili. Per es: encoding delle stringhe, layout delle informazioni, eccetera.
In attesa di I/O, il thread è bloccato; la sua prosecuzione dipende da un evento esterno. Va gestita la corretta attenzione ad altri eventi che potrebbero essere di interesse per l'applicazione (interazione con l'utente, segnali di interruzione, eccetera).

L'oggetto *Datagram* modella il pacchetto UDP corrispondente ad un messaggio singolo inviato ad uno o più destinatari. Al contrario del Socket, non c'è connessione: l'invio è asincrono e non blocca il mittente.
Il ricevente si deve mettere in ascolto, attendendo la ricezione di un messaggio. Per le caratteristiche intrinseche del protocollo, i messaggi di questo tipo hanno una affidabilità minore.

Per offrire un'alternativa alla gestione manuale delle attese su Socket e Datagram, è stata introdotta l'astrazione del *Channel*. In questa astrazione, una volta scelto il canale di comunicazione, si può predisporre l'azione da eseguire in risposta ad un evento di I/O, in modo da delegare al componente la gestione delle risorse in attesa. Le chiamate sono completamente asincrone, e richiedono di propagare manualmente un oggetto di contesto per identificare e legare fra loro eventi che riguardano la medesima conversazione.

Le difficoltà insite nel paradigma distribuito sono a volte ingannevoli, e portano a fare ipotesi che sono in realtà pesantemente false. La pratica ha suggerito otto di queste ipotesi su cui è molto comune cadere in errore, definite *fallacies*:

1) *The network is reliable.*
2) *Latency is zero.*
3) *Bandwidth is infinite.*
4) *The network is secure.*
5) *Topology doesn’t change.*
6) *There is one administrator.*
7) *Transport cost is zero.*

Scritto da Gabriel

8) *The network is homogeneous.*
9) *(Sarebbero 8 ma la cito) Time is ubiquitous*

Un *framework* è un insieme di componenti, tecnologie e scelte metodologiche che propone un insieme di soluzioni ad un determinato campo di problemi. Nel caso dello sviluppo di applicazioni distribuite, un framework proporrà un modello di comunicazione più comodo delle primitive di base, ed un insieme di regole ed interfacce per interagirci.
Usare un framework è *vantaggioso* perché:

1) si può usare una infrastruttura testata e realizzata con competenza
2) si possono usare soluzioni già fatte a problemi comuni
3) possiamo concentrarci sul codice che risolve il nostro problema

Usare un framework è *svantaggioso* perché:

1) gli sviluppatori potrebbero non avere le nostre stesse priorità
2) il nostro caso d'uso non è ben supportato
3) possiamo incontrare errori e guasti per noi imprevedibili
4) l'evoluzione del framework può richiederci costi di sviluppo non controllabili

Le stesse motivazioni che hanno portato alla ricerca della distribuzione per l'esecuzione di un compito sono valide anche per la conservazione di dati. Vogliono garantire la disponibilità anche in caso di guasto parziale, gestendo una mole maggiore delle capacità di un solo nodo e mantenendo l'accessibilità da più posizioni geografiche. Nel caso in cui un sistema conservi il suo stato interno suddiviso fra i nodi che lo compongono, nasce il problema di mantenere questa rappresentazione coerente nel suo complesso. Tale problema è detto *problema del Consenso*.
Il problema richiede l'uso di particolari algoritmi che garantiscano l'affidabilità del consenso del sistema anche in presenza di alcune classi di errori o di guasti. In letteratura, i più diffusi sono *Paxos* e *Raft*.
Il *Teorema CAP/CAP Theorem* fornisce un limite alle funzionalità che un sistema distribuito può realizzare in caso di un certo tipo di guasti, in particolare descrivendo:

1) *C: Consistency*
2) *A: Availability*
3) *P: (Network) partition*

Il teorema afferma che in caso di suddivisione della rete in due parti, il sistema deve scegliere se: mantenere la consistenza, sacrificando la disponibilità continuare a rispondere, accettando i conflitti.
Una estensione considera anche il funzionamento nominale:

1) *P: in case of partitioning*
2) *A: (choose between) availability*
3) *C: and consistency*
4) *E: else, when normal, choose between*
5) *L: latency*
6) *C: and consistency*

Per gestire il caso di modelli dati in cui più nodi devono poter contemporaneamente scrivere, è possibile utilizzare delle strutture dati *CRDT*. Una struttura dati *Conflict-Free Replicated Data-Type* garantisce di avere sempre un modo per riconciliare scritture avvenute contemporaneamente su versioni diverse delle informazioni, in modo da poter sempre convergere su di un valore che le include tutte.
Le strutture dati CRDT sono abbastanza facili da usare, ma molto limitate nelle possibilità semantiche e a volte molto costose in termini di spazio e complessità di calcolo.

La *reattività* è il paradigma di programmazione che ha come obiettivo la realizzazione di sistemi fortemente asincroni ma comunque semplici da programmare, con una particolare enfasi nel mantenere molto bassa la latenza alla risposta a stimoli esterni.
Per sfruttare la semplicità del modello di elaborazione dello Stream con maggiore potenza espressiva, le *Reactive Extensions* definiscono un modello semantico più preciso.

Vengono definiti quattro componenti astratti, ed un ampio insieme di operatori che permettono di usarli:

1) *Observable*
2) *Scheduler*
3) *Subscriber*
4) *Subject*

Ogni componente è attentamente definito sia nel funzionamento sia nel comportamento asincrono e concorrente. Questa modalità di utilizzo è di primaria importanza nella costruzione della semantica del sistema. Il risultato è un metodo che si adatta a vari linguaggi e porta in essi la stessa potenza espressiva e facilità d'uso.

Costruendo sull'esperienza delle Reactive Extensions, i *Reactive Streams* ampliano la gamma di garanzie disponibili. La pipeline di elaborazione può gestire la *back-pressure*, ovvero opporre resistenza alla velocità dei dati in ingresso. La semantica che viene aggiunta prevede che un componente possa comunicare al precedente quanti dati è in grado di elaborare. Questo consente di stabilizzare la capacità della pipeline di elaborazione e garantire che nessun componente venga soverchiato dal flusso di dati in ingresso, ottenendo maggiore affidabilità complessiva. Inoltre, la possibilità che i componenti di elaborazione si trovino su nodi differenti è esplicitamente gestita, allo scopo di ricercare scalabilità ed efficienza ai guasti.

Nella ricerca di una affidabilità estremamente elevata, viene ideato un paradigma completamente nuovo, basato su di una architettura estremamente concorrente, trasparentemente distribuita e tollerante ai guasti. Il paradigma si basa sulla modellazione della singola esecuzione come un *Attore*, con una semantica molto ben definita delle operazioni che un tale oggetto può fare, e delle garanzie che ha a disposizione durante l'esecuzione. Un Attore comunica con l'esterno esclusivamente tramite messaggi indirizzati ad altri attori. Al suo interno, è completamente *thread-safe*: ogni risposta ad un messaggio verrà elaborata da un unico thread.

Nel rispondere ad un messaggio un attore può unicamente:

1) cambiare il proprio stato interno
2) creare nuovi attori
3) inviare messaggi
4) cambiare il proprio comportamento per il prossimo messaggio

Un Attore può fallire in qualsiasi momento. L'Attore che l'ha creato può ricevere un messaggio e decidere che strategia usare per garantire il proseguo del funzionamento del sistema.
Con questo tipo di primitiva è possibile realizzare sistemi estremamente performanti e resilienti, dotati di caratteristiche di scalabilità quasi lineare. Programmare un attore è tuttavia molto complesso, e richiede una formazione ed una mentalità ad hoc, data la distanza dal normale modo di scrivere codice.

In conclusione

Motivazioni economiche e tecnologiche ci portano a dover eseguire il nostro codice in modo concorrente o distribuito. Per ottenere delle soluzioni corrette è necessario approcciare correttamente queste situazioni. Gli strumenti di base sono molto legati allo specifico problema, ma spesso difficili da usare correttamente. È di norma meglio cercare una astrazione di livello superiore, verificando quali garanzie sono mantenute. L'ecosistema attorno a Java e alla JVM è in grado di coprire una vastissima gamma di problemi, in differenti modalità di esecuzione, con semplicità d'uso e grande compatibilità con ogni piattaforma di esecuzione. La piattaforma JVM è molto adatta, e molto usata, per lo studio e la realizzazione di strumenti innovativi e di grande efficacia in ogni paradigma di programmazione.