Paradigma di programmazione

Tecnica o insieme di tecniche per affrontare una classe di problemi.

Java

Il compilatore Java produce un file con estensione `.class` per ogni classe ottenuta dal codice. Un parametro molto importante per la JVM è il `CLASSPATH`, ovvero l'elenco dei posti dove cercare una classe.\ Il comando `java` avvia la JVM ed esegue il bytecode contenuto nel `CLASSPATH`.

Maven

Per Maven il progetto è descritto da un POM, Project Object Model, dove vengono elencati: nome e metadati del progetto, dipendenze, plugin e loro configurazioni. Ogni libreria può essere aggiunta a un progetto indicandolo secondo delle coordinate: `gruppo:artefatto:versione` attraverso esse maven è in grado di procurarsi il `jar` del componente da una repository remota.

Gradle

Alternativa a Maven che punta a migliorarne alcuni limiti: non più xml ma un linguaggio di programmazione (Groovy) per definire il progetto. Non si ha più una struttura fissa ma task che dipendono l'uno dall'altro.

Classi

Le variabili statiche vengono allocate e inizializzate nel momento in cui la classe viene caricata dal `ClassLoader` (carica dinamicamente le classi nella JVM a runtime).

Visibilità delle variabili

`public`, `protected`, default (lette e scritte da classi nello stesso package), `private`.

`Throwable`

Tutte le eccezioni derivano dalla classe `Throwable`, che ha due sottoclassi:\ `Exception`: gli errori nonostante i quali il programma dovrebbe essere in grado di proseguire. Una sua particolare sottoclasse è `RuntimeException`, che viene lanciata direttamente dalla JVM e può avvenire durante la valutazione di espressioni.\ `Error`: gli errori dai quali non è in grado di proseguire.

> ⓘ **Unchecked vs checked exceptions**
>
> Eccezioni derivate da `RuntimeException` e `Error` sono dette *unchecked* e non necessitano dichiarazione nella definizione di un meotodo. Tutte le altre, discendenti da `Exception` o `Throwable` direttamente sono dette *checked* e devono essere dichiarate.

Nested classes

Hanno un modificatore di visibilità come gli altri campi di una classe.\
Classi interne statiche sono chiamate ***static nested classes***, che non hanno accesso privilegiato ai membri della classe ospite, quindi è come se fosse un'altra classe del package e ha un nome prefissato da quello della classe ospite. Sono spesso usate in *design pattern*.
Le classi interne non statiche sono dette ***inner classes*** e sono una parte dello stato di un oggetto del tipo ospite, quindi ha lo stesso ciclo di vita e ha un riferimento privilegiato all'oggetto ospitante. Non può dichiarare membri `static` ma solo membri di istanza.

Inizializzatori

In una classe possono essere contenuti anche blocchi di codice anonimi, eseguiti all'inizializzazione di una classe o un oggetto. I blocchi di inizializzazione `static` vengono eseguiti al caricamento della classe.\ Quelli senza indicazioni sono eseguiti (in ordine lessicale) *dopo* il supercostruttore ma *prima* di qualsiasi costruttore.

> **Il codice nell'inizializzatore deve essere veloce e non può fallire.**

```
static String s = "foo";
static int len;

static {
        len = s.length();
}
```

> **Ordine di inizializzazione**
>
> 1. Inizializzatori statici
> 2. Costruttori delle superclassi
> 3. Inizializzatori della classe
> 4. Costruttore della classe

Ereditarietà

In Java è presente un meccanismo di ereditarietà singola, una classe può avere una sola superclasse, quindi ha evitato il **diamond problem**. Per costruzione, tutti i metodi Java sono *virtual* (nel senso C++), ovvero il codice che realmente viene eseguito alla chiamata di un metodo è noto con certezza solo a runtime.\ Una classe `final` non può essere usata come superclasse.\ Una classe `abstract` deve essere usata come superclasse (non direttamente istanziabile).\ Una classe dichiarata `sealed` può elencare i tipi che la possono estendere:

```
public abstract sealed class Shape permits com.example.Circle, com.example.Rectangle
{}
```

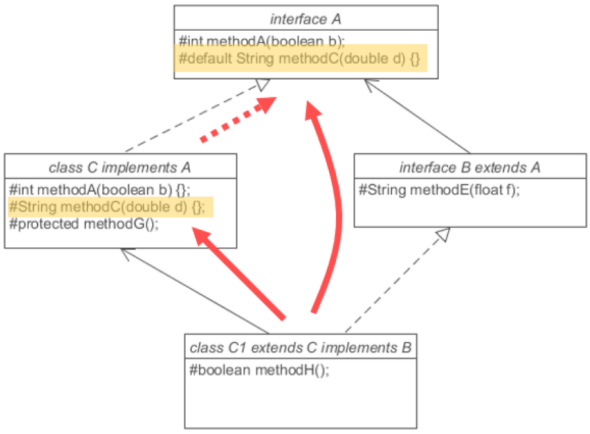
Interfacce

L'ereditarietà è limitante perché non permette la *composizione*, ovvero prendere le parti necessarie da più classi.\ Un'interfaccia dichiara le caratteristiche di un Tipo, senza fornire una sua implementazione (*contratto* a cui una classe può aderire, che deve fornire un'implementazione di tutti i metodi). Un'interfaccia può essere estesa solo da un'altra interfaccia e può avere visibilità pubblica o di package e tutti i suoi membri sono pubblici. Solo dopo Java 8 un'interfaccia può contenere metodi di default. Dopo Java 9 metodi privati. Può contenere:

- Dichiarazioni di costanti
- Metodi astratti e metodi statici

Default methods

Metodi implementati che si comportano come se fossero metodi delle superclassi. Si usa la keyword `default`. Quindi, per mezzo dei *default methods* un'interfaccia può essere modificata aggiungendo nuovi metodi, senza che le sue implementazioni debbano essere toccate. Se il metodo non è gestito dalla classe, viene usato quello dichiarato nell'interfaccia. Tuttavia, questo porta al ripresentarsi del *diamond problem*:



Se la gerarchia porta ad ambiguità nella selezione del metodo, esso viene rilevato al momento della compilazione.

Annotazioni

Il compilatore può eseguire degli *Annotation Processors* che sono interfacce il cui nome inizia per `@`. Durante la compilazione, possono produrre nuovi file a partire dalle annotazioni stesse.

Generics

Si possono esprimere vincoli sul parametro:

```
interface SortableList<T extends Comparable<T>>{...}
```

Si possono anche non esprimere vincolo sui parametri, ma solo sul tipo parametrizzato. In questo modo non viene condizionato il contenuto della collezione: `Collection<?>`.\ Un tipo primitivo non

può essere usato come parametro di tipo. È presente il concetto di *autoboxing*, in cui il tipo primitivo viene converito nella corrispettiva classe. Un array invece è un oggetto.

Enum

Non è possibile istanziare una classe `enum`, ma solo usare i suoi valori. Se è una classe interna, è implicitamente `static`.\ Un'`enum` può dichiarare variabili, metodi o implementare interfacce. Se nessun elemento dichiara un'implementazione specifica, l'`enum` è `final`.

Records

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

È una sorta di *struct*. I record sono immutabili (`final`), quindi non può essere astratto né esteso. Un record ottiene automaticamente getter (senza prefisso get. Non hanno setter essendo `final`), costruttore con tutti gli elementi, `equals`, `hashCode`, `toString`. Si possono ridefinire i metodi di un record generati automaticamente.

Try-with-resources

```
try(BufferedReader br = new BufferedReader(new FileReader(path))){
        return br.readLine();
}
```

Permette di dichiarare variabili che devono implementare l'interfaccia `AutoClosable` e vengono automaticamente chiuse al termine.

Modificatori

Campi

- `final` (il valore non può essere modificato dopo l'assegnamento, che deve necessariamente avvenire alla costruzione)
- `transient` la variabile va ignorata al momento della serializzazione
- `volatile` la variabile ha un comportamento particolare in relazione all'accesso concorrente.

Classi

`strictfp` il codice della classe usa semantica Floating Point restrittiva. Ovvero tutti i membri della classe devono essere omogenei
(o tutti float o tutti double).

Metodi

- `abstract` deve essere implementato da una classe di estensione
- `final` non può essere reimplementato da una classe di estensione
- `native` implementato da una libreria nativa
- `strictfp`

Libreria Standard

L'istruzione `switch` può essere scritta anche come lambda expression. Per ritornare valori si usa la keyword `yield` e non c'è *fall-through*. Nella classe `Collection` sono presenti tutti i metodi base per gestire una collezione, come `sort`...

Set

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

Un `SortedSet` è un insieme su cui è definito un ordine totale: è possibile enumerarlo secondo tale ordine e individuare inizio e fine dell'insieme. Un `NavigableSet` è un insieme ordinato su cui è possibile muoversi sfruttando l'ordine, cercando ad esempio l'elemento minore o maggiore di un elemento dato.

Dequeue

`Dequeue` rappresenta una Double Ended Queue, ovvero una struttura dati da cui è possibile togliere e aggiungere elementi dall'inizio o dalla fine. Quindi è sia FIFO che LIFO. Essa possiede due set di metodi differenti, a seconda del comportamento richiesto in caso di impossibilità dell'azione richiesta.

- `add, remove, get` lanciano un'eccezione
- `offer, poll, peek` ritornano un valore speciale (`null`, `false`).

Map

Implementazione	Caratteristiche
`HashMap`	Chiavi distinte per `hashCode`
`TreeMap`	Chiavi ordinate
`HashTable`	Legacy
`EnumMap`	Specifica per chiavi `enum`
`WeakHashMap`	Chiavi "deboli", se oggeto non riferito il Garbage Collector lo dealloca
`IdentityHashMap`	Specifica basata sull'identità

Come per il `Set` esistono le corrispettive `SortedMap` e `NavigableMap`.

Stream

Uno `stream` è una sequenza di elementi, non necessariamente finita. Le operazioni sugli `stream` vengono composta in sequenza in una *pipeline*, fino ad arrivare ad un'operazione detta **terminale** (ovvero continuo ad avere sempre uno stream, è *lazy*), che produce il risultato. Gli `stream` sono

utili perché consentono di descrivere il significato dell'elaborazione, invece che il metodo, permettendo l'ottimizzazione dell'esecuzione. Nessuna operazione viene eseguita finché non viene richiamata l'operazione terminale.\ Le operazioni sulla pipeline **non devono cambiare gli elementi dello stream** (eventualmente trasformarli) e si dividono in **stateless** e **stateful**. Esempi:

Stateless	Stateful	
`filter`	`distinct`	
`map`	`concat`	Concatena due stream
`drop/takeWhile`	`limit`	Tronca lo stream
	`skip`	Salta l'inizio dello stream
	`sorted`	

Terminal	
`all/any/noneMatch`	`forEach/Ordered`
`collect`	`min/max`
`findAny/findFirst`	`reduce`
`flatMap`	

`generate` produce uno stream a partire da un `Supplier` (si può generare uno stream ipoteticamente infinito, con elementi generati da Supplier).
`iterate` produce uno stream applicando una funzione a partire da un seme (ad esempio se il seme 1 e la funzione raddoppia).\ *Con lo stream ci sono più informazioni per ottimizzare l'esecuzione.*

Programmazione concorrente

Teoria e tecniche per la gestione di più processi sulla stessa macchina che operano contemporaneamente condividendo le risorse disponibili.\ *L'obiettivo del sistema operativo è garantire l'utilizzo efficace e paritario delle risorse, non favorirne l'uso contemporaneo.*

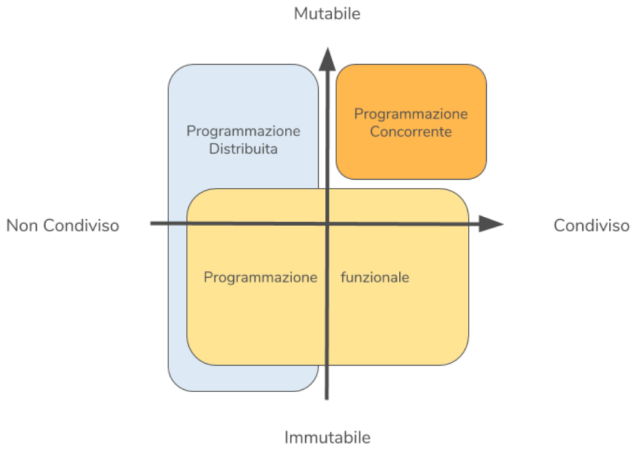
Processo

In UNIX il principale concetto di gestione delle attività è il processo, che descrive un programma in esecuzione e tutte le risorse che gli sono dedicate (memoria, canali di I/O, interrupt e segnali, stato della CPU). Normalmente i processi non condividono risorse, ma tramite il SO può avvenire.\ Il cambio di contesto è molto costoso, quindi è stato ideato il concetto di **Thread**, per gestire più linee di esecuzione all'interno dello stesso processo. I Thread condividono le risorse di uno stesso processo. Questo però riporta all'applicazione il problema di gestione di accesso contemporaneo alle risorse.

Confronto tra paradigmi di programmazione

- La **programmazione distribuita** implica la comunicazione fra entità che non possono avere uno stato condiviso.
- La **programmazione funzionale** tratta preferibilmente dati immutabili. Lo stato può essere distinto o (specie se immutabile) condiviso.

- Nella **programmazione concorrente** lo stato è mutabile e condiviso, quindi l'accesso e l'intervento su di esso va coordinato e gestito.
- La **programmazione ad oggetti** riguarda tutti e 4 gli stati.



Problemi della programmazione concorrente

- Non-determinismo: non dipende più dall'algoritmo ma da quando vengono assegnate le risorse
- *Starvation*: un thread che non riceve abbastanza risorse non può fare il suo lavoro.
- *Race conditions*: se più thread competono per le stesse risorse, il loro ordine di esecuzione può essere rilevante per il risultato.
- *Deadlock*: due thread attendono ciascuno la risorsa che ha già preso l'altro e nessuno dei due può più proseguire.

Condizioni di Coffman

Condizioni necessarie affinché possa avvenire una deadlock. Rimuovere anche solo una delle condizioni rende impossibile entrare in una deadlock.

1. **Mutua esclusione**: la risorsa è utilizzabile da un solo thread alla volta (algoritmi lock-free, sono in grado di gestire una risorsa non acquisibile).
2. Attesa della risorsa e trattenimento della stessa (***hold and wait***): cioé se non c'è modo di *prenotare* l'acquisizione di una risorsa, il processo passerà tutto il tempo verificare se si è liberata. I processi che detengono risorse possono chiederne altre (assegnazione delle risorse transazionale, ovvero poter acquisire tutte le risorse necessarie).
3. Assenza di prerilascio (***no preemption***): le risorse possono essere rilasciate solo dal processo che le detiene, il rilascio non può essere forzato. (algoritmi lock-free o *optimistic concurrency control*, si tenta di fare un'operazione e poi si verifica se la risorsa è rimasta al processo per tutto il tempo. Se non è vero dovrò riprovare a farlo).
4. **Attesa circolare** (imporre un ordinamento dell'acquisizione delle risorse)

Tipologie di concorrenza

- **Collaborativa (strutture: co-routines)** I programmi devono esplicitamente cedere il controllo ad intervalli regolari (usato nei sistemi *embedded*).
- **Pre-emptive (processi, thread)** Il sistema operativo è in grado di interrompere l'esecuzione di un programma e sottrargli il controllo delle risorse per affidarle al programma seguente (usato

nei SO moderni).

- **Real-time (processi, thread)** Il sistema operativo garantisce prestazioni precise e prefissate nella suddivisione delle risorse fra i programmi (usato per applicazioni molto particolari).
- **Event driven/async (future, events, streams)** I programmi dichiarano le operazioni che vanno eseguite e lasciano all'ambiente di esecuzione la decisione di quanto eseguirle e come assegnare le risposte.

Metodi della classe Thread

Allocare un nuovo oggetto Thread:

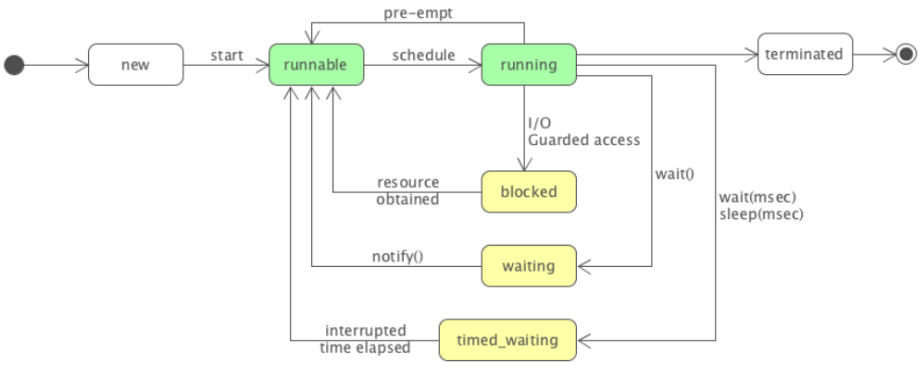
```
public Thread(Runnable target)
public Thread(ThreadGroup group, // raggruppare sicurezza/priority per assegnazione risorse
                          Runnable target,
                          String name,
                          long stackSize);
```

Metodi:

```
void start(); // si comporta come fork
public String getName();
public boolean isAlive();
public void run();
public static Thread currentThread();
public static void sleep(long millis) throws InterruptedException
public void interrupt(); // interrompe questo thread
```

Se il Thread è stato costruito usando un oggetto `Runnable` (che modella un compito da cui non ci si aspetta un risultato) separato, invocando `run()` viene chiamato il metodo `run` di quell'oggetto, altrimenti non fa nulla e ritorna.

Stati di un Thread



Stati di un thread

- **Runnable** Il thread non sta consumando risorse, ma è pronto a farlo
- **Blocked** Il Thread ha richiesto accesso ad una risorsa monitorata (es: canale di I/O) e sta aspettando la disponibilità di dati.

- **Waiting** Il Thread è in attesa di una risorsa protetta da un lock chiamando `wait` e sta aspettando il suo turno.
- **Timed Waiting** Il Thread si è posto in attesa per un determinato periodo di tempo (es con `sleep()`)
- **Terminated** Il metodo `run()` è completato correttamente o meno e il Thread ha concluso il lavoro. Per settare l'handler invocato quando questo thread termina dovuto ad un'eccezione non gestita:

  ```
  public void setUncaughtExceptionHandler(Thread.uncaughtExceptionHandler e);
  ```

Executors

Creare un nuovo Thread per ogni operazione può essere costoso. La soluzione è cedere il controllo al sistema (gestire automaticamente i Thread).

```
public interface Executor
```

Prende in input come gestire lo scheduling, l'accodamento, l'allocazione di risorse... Dopodiché tramite il metodo `void execute(Runnable command)` prende in input la task da eseguire.

Thread pool

Un concetto ricorrente nei thread pool è il riutilizzo dei thread più volte per diversi task durante il suo ciclo di vita. Questo diminuisce l'overhead dovuto alla creazione dei thread e incrementa le prestazioni del programma che sfrutta il thread pool.

Costruisce al massimo 4 thread. Se vengono assegnate più di 4 attività, dovranno attenere il termine.

```
Executor executor = Executor.newFixedThreadPool(4);
var threads = Stream.generate(new ThreadSupplier());
threads.limit(10).forEach((t) -> executor.execute(t));
```

Tipo	Strutture
`CachedThreadPool`	Riusa thread già creati, ne crea nuovi se necessario (non ha limite superiore alla cache, la sua strategia è di mantenere la coda vuota)
`FixedThreadPool`	Riusa un insieme di thread di dimensione fissa
`ScheduledThreadPool`	Esegue i compiti con una temporizzazione
`SingleThreadExecutor`	Usa un solo thread per tutti i compiti
`ForkJoinPool`	Punta ad usare tutti i processori disponibili

Callable

I `Runnable` sono dei blocchi di codice privi di risultato. L'interfaccia `Callable` permette di definire task che producano un risultato (e potrebbe lanciare un'eccezione).

```
@FunctionalInterface
public interface Callable<V>{
        V call() throws Exception;
}
```

Executor Service

Un semplice `Executor` non esegue `Callable`, è necessario scegliere un `ExecutorService`.

```
public interface ExecutorService extends Executor {
        <T> Future<T> submit(Callable<T> task);
}
```

`submit` permette di inviare una task per l'esecuzione e ritorna un `Future` che rappresenta il risultato in pending della task, che prima o poi produrrà un risultato.

Con a disposizione una lista di `Callable`, un `ExecutorService` permette di:

- Ottenere un risultato di un `Future` che ha terminato (non necessariamente il primo, ma uno dei primi che termina senza eccezioni), tramite il metodo `invokeAny`:

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks);
```

- Ottenere una lista di `Future`, nel momento in cui sono tutti completati, tramite il metodo `invokeAll` che ritorna una lista di Future:

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks);
```

Future

Il metodo `T get()` nella classe `Future` attende che la computazione sia completata, poi restituisce il risultato.
Un `ExecutorService` rimane sempre in attesa di nuovi compiti da eseguire, impedendo alla JVM di terminare. Per fare ciò è necessario chiamare esplicitamente il metodo `shutdown()` (non vengono accettati nuovi task e tutti i task in coda vengono terminati).
`awaitTermination` blocca fino a quando tutte le task sono completate dopo una richiesta di shutdown, dopo che è scaduto il timeout oppure quando il thread corrente viene interrotto.

```
boolean awaitTermination(long timeout, TimeUnit unit); //false quando attività
terminate
```

Il metodo `shutdownNow` tenta di fermare tutte le task in esecuzione e ritorna una lista di quelle in attesa.

```
List<Runnable> shutdownNow();
```

Sincronizzazione

> ⓘ **Sezione critica**
>
> Parte del codice in cui vengono acceduti i dati condivisi. Se più thread si trovano nella sezione critica, potrebbero verificarsi errori/incoerenze di dati.

`synchronized` può essere utilizzato su un blocco di istruzioni oppure su un metodo.
Tutti i blocchi sincronizzati di un oggetto condividono lo stesso `monitor lock` (o `intrinsic lock`), che interviene nel momento in cui viene usata la parola chiave `synchronized` per controllare l'accesso al suo stato interno.

Usando `synchronized` stabiliamo una relazione di **happens-before** tra l'azione di rilascio del lock e ogni successiva acquisizione dello stesso (è una garanzia forse fornita dal compilatore). La relazione happens-before impedisce al compilatore di ottimizzare il codice riordinandone le istruzioni.

```
synchronized(that){ // synchronized senza (that) è equivalente a non averlo o avere (this)
        /* permette di rendere il blocco di codice sincronizzato sul monitor
        dell'oggetto ritornato dall'espressione that */
}
```

Tutti i monitor sono `reentrant`: un Thread può acquisire lo stesso monitor più volte (quindi eseguire più metodi synchronized dello stesso oggetto) senza entrare in *deadlock* con se stesso.

Wait

Un'alternativa a `synchronized` è la gestione esplicita del monitor di un oggetto.

```
// il thread corrente attende fino a quando un altro thread non chiama notify() o notifyAll()
void wait() throws InterruptedException;
void notify(); // riavvia un singolo thread che sta aspettando sul monitor
void notifyAll();
```

Per poter operare sul monitor dell'oggetto, il thread deve poter avere a disposizione l'oggetto stesso. Può farlo:

1. Eseguendo un metodo/blocco `synchronized` dell'oggetto
2. Eseguendo un metodo `synchronized static` (di una classe)

Producer/Consumer

`synchronized` crea un blocco implicito, `wait()` ci costringe a gestire lo stato del blocco. Si potrebbe voler avere un controllo esplicito delle condizioni di blocco/sblocco della sezione critica:

```
public interface Lock
void lock(); // acquisisce il lock, blocca se non è disponibile
void unlock(); // rilascia il lock
boolean tryLock(); // non blocca, ritorna true solo se è libero al momento
dell'invocazione
```

Pattern architetturale che modella un insieme di thread divisi in due gruppi: *producers* (thread che producono dati da elaborare) e *consumers* (elaborano i dati prodotti). Con un `Lock` possiamo controllare manualmente l'acceso alla sezione critica (il `Consumer` ammette solo un Thread alla volta nella sezione critica, quindi il `Lock`, implicitamente gestisce la coda dei produttori in attesa).

Alcune implementazioni, come il `ReentrantLock`, permettono di chiedere che l'implementazione sia fair (il Thread che riceve il lock è quello che ha aspettato di più). Quindi si riduce il rischio di starvation a prezzo di una prestazione inferiore (dev'essere gestita una lista ordinata).
Un `ReentrantLock` ha caratteristiche equivalenti a un `implicit lock` ma può essere controllato manualmente.

Conditions

A volte non tutti i Thread che cercano di acquisire un lock sono uguali, possono avere semantiche diverse ed essere segnalati in condizioni differenti.\ Con `Condition` è possibile gestire su un solo lock più condizioni di attesa distinte e a seconda della condizione che si è verificata, avviare un thread di un gruppo o di un altro. Su una `Condition` un thread si può mettere in attesa con `await` e può essere risvegliato con `signal` o `signalAll`.

Semaphores

Per controllare l'accesso ad un insieme omogeneo di risorse si usa un semaforo, che è simile ad un lock ma tiene un conteggio invece che un semplice stato libero/occupato.

```
public Semaphore(int permits, boolean fair);
```

Se è inizializzato come `fair`, l'ordinamento dei Thread in attesa è garantito come FIFO, altrimenti non è garantito. Ad ogni acquisizione (`acquire` o `acquire(int permits)`) il numero di `permits` diminuisce, ad ogni rilascio (`release` o `release(int permits)`) aumenta.
`acquire(int permits)` blocca fino a quando non sono disponibili quel numero di permessi.
Il valore iniziale del semaforo non è un limite, può essere superato e andare in negativo (sta all'utilizzatore gestirlo). A differenza di un `lock`, un semaforo può essere rilasciato da un thread diverso da quello che lo ha acquisito. La maggior parte dei metodi di `Semaphore`:

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

```
// prova ad acquisire uno o più permessi solo se disponibili
public boolean tryAcquire();
public boolean tryAcquire(int permits, long timeout, TimeUnit unit);
```

⚠ **`tryAcquire` è in grado di violare la fairness del semaforo.**

```
// ridurre il num di permessi dispoibili senza acquisirne
protected void reducePermits(int reduction);
```

Dati *Thread-Safe*

Il problema di condividere l'accesso ai dati, oltre alla *deadlock*, è anche quello della correttezza del risultato. Se abbiamo necessità di condividere dati tra più thread, abbiamo bisogno di strutture *thread-safe*. Se una struttura dati viene modificata da più thread, nel migliore dei casi lancia una `java.util.ConcurrentModificationException`, nel caso intermedio lo stato diventa inconsistente, nel peggiore dei casi si ottiene un'altra eccezione.

Atomic variables

Classi del package `java.concurrent.atomic` (`AtomicInteger`, `AtomicReference` per gli `Object` e include anche i tipi per gli array), garantiscono che la modifica del valore che contengono sia atomica e *thread-safe* e che la modifica non blocchi il thread che la sta eseguendo. Quasi sempre la funzionalità richiede la disponibilità del supporto hardware attraverso istruzioni CAS (*Compare-And-Swap*).

```
public final boolean compareAndSet(long expect, long update);
```

Confronta il valore e lo aggiorna, ritornando falso se non corrisponde con il valore aspettato.

```
public class AtomicMarkableReference<V>; // tenere un riferimento a un oggetto e
marcarlo come "usato" tramite un mark bit
public class AtomicStampedReference<V>; // associare un timestamp all'oggetto (in
modo che si possa controllare se il valore è cambiato inaspettatamente dall'ultima
modifica).
```

`volatile`

Indica che una variabile deve essere sempre letta dalla memoria principale e non da cache intermedie. Thread che sono su CPU diverse vedrebbero valori diversi della stessa variabile, proveniente dalla cache: problema della ***visibilità del valore scritto***.
Un altro caso in cui questo problema si presenta è nell'interazione con codice nativo interfacciato via Java Native Interface (es: driver scritti in C++) che riporta nella memoria principale una misura che cambia continuamente.
In questo modo si garantisce la visibilità dell'ultima scrittura. Si stabilisce una relazione di *happens-before* tra una scrittura e qualunque lettura successiva di quel campo.

> ⚠ **La garanzia fornita da `volatile` è utile alla correttezza del programma se nessun thread scrive nella variabile un valore che dipende dal valore appena letto dalla stessa variabile.**

Concurrent Data Structures

Più efficienti della versione sincronizzata.

Map

`ConcurrentMap` estende `Map` e aggiunte garanzie di atomicità e ordinamento delle operazioni. Ha dei metodi:

```
V putIfAbsent(K key, V value); // dà garanzia di atomicità
V replace(K key, V value); // replace solo se è presente
V replace(K key, V oldValue, V newValue); // replace solo se è oldValue
```

`ConcurrentHashMap` offre metodi come `reduce, search,` (ricerca parallela che ritorna uno dei primi valori trovati) `foreach` che possono operare su tutte le chiavi suddividendo il lavoro in più thread. Queste funzioni non sono atomiche rispetto ad altre modifiche effettuate nel frattempo, ma garantisce una relazione di *happens-before*: se viene effettuata una modifica, l'iterazione non va in errore e legge quella modifica se avviene prima.

Reduce

Funzione che implementa l'algoritmo *map-reduce*, prende come parametro un trasformatore, che da una coppia key-value ritorna un valore e un'operazione associativa su questo risultato applicata a tutti i valori ritornati dal trasformatore, in modo da ottenere un valore solo. In questo caso la mappatura può essere fatta concorrentemente tra più thread.
Prende in input anche `parallelismThreshold`, che permette di stabilire di utilizzare il parallelismo solo se la dimensione della mappa supera una certa soglia.

> ⚠ **Warning**
>
> Le funzioni usate nei metodi di trasformazione delle mappe devono:
>
> - Non dipendere dall'ordinamento
> - Non dipendere da uno stato condiviso durante il calcolo.
>   I metodi diversi da `forEach` non devono avere effetti collaterali.

Queue

`BlockingQueue` aggiunge alla `Queue` metodi con cui è possibile scegliere la semantica dell'operazione di accodamento e prelievo.

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

Metodo	Risultato negativo
`poll()`	`null`
`take()`	attesa
`poll(time, unit)`	attesa limitata

Lettura

Metodo	Risultato negativo
`element()`	eccezione
`peek()`	`null`

```
int drainTo(Collection<? super E> c); // Rimuove tutti gli elementi disponibili da questa queue e li aggiunge alla collezione c. Ritorna il numero di elementi trasferiti. È più efficiente che fare il poll dei singoli elementi, perché non vengono fatti controlli sulla concorrenza
```

La `BlockingQueue` può essere utilizzata per implementare sistemi Produttore-Consumatore.

Thread Local Variables

Potremmo volere che la stessa variabile abbia un valore indipendente e separato per ciascun Thread.

```
public class ThreadLocal<T>
```

Quindi una variabile `ThreadLocal` esiste in una copia differente e indipendente per ciascun Thread che attraversa la sua dichiarazione.

```
static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier);
public T get(); // valore corrispondente al thread che fa quella chiamata (risultato diverso in base al Thread)
public void remove(); // rimuove il valore del thread corrente
void set(T value);
T initialValue(); // ritorna il valore iniziale
```

Parallel Streams

Uno stream rappresenta l'iterazione su un insieme di cardinalità non nota, potenzialmente infinita.

Differenze tra Stream e Collezioni

- Se il codice è stato scritto come operazioni su stream, non è necessario cambiare il codice in caso diventi concorrente
- Nelle Collezioni, a seconda dell'algoritmo si sceglie la collezione (ognuna ha performance di accesso ai contenuti diversa e l'algoritmo che la utilizza è estraneo a questa considerazione). Con gli stream invece si definisce l'algoritmo che viene applicato sui suoi elementi (ma essi non possono essere acceduti direttamente). Lo

stream ha controllo sull'algoritmo e può prendere decisioni su come eseguire questo algoritmo. In altre parole:

Le collezioni non sanno quale algoritmo verrà applicato, mentre gli stream si

Stream Flags

La sorgente di uno Stream può dichiarare alcuni flag, che gli operatori intermedi possono verificare e l'operazione terminale usa per prendere decisioni sull'esecuzione.

- `CONCURRENT` parallelizzabile (eseguito in più thread paralleli)
- `DISTINCT` garantisce che elementi siano distinti rispetto ad `equals()`
- `IMMUTABLE` immutabile durante il consumo
- `NONNULL` elementi non nulli
- `ORDERED` è garantito che gli elementi vengano ritornati nell'ordine in cui sono stati emessi (anche dopo un'esecuzione parallela)
- `SIZED` dimensione nota
- `SORTED` mantiene gli elementi in un ordinamento naturale o definito da un `Comparator` (`ORDERED` è l'ordine di apparizione)
- `SUBSIZED` può fornire dei sottoinsiemi dimensionati per l'esecuzione parallela
  L'operazioni terminale ha piena visibilità su quali sono le caratteristiche dello stream e può prendere decisioni.

`BaseStream`

Interfaccia di base estesa da tutti gli stream, che supportano operazioni sequenziali e parallele.

```
interface BaseStream<T, S extends BaseStream<T, S>>
S parallel();   // ottenere una versione parallela dello stream
S sequential(); // ritorna un equivalente sequenziale
```

> ***short-circuing***
>
> Operazioni che possono interrompere l'esecuzione della pipeline prima dell'esame di tutti gli elementi, come `anyMatch`, `limit` oppure `findAny`

Operazioni ***stateful***

Operazioni che hanno bisogno di consumare tutto o gran parte dell'input per poter emettere l'output, mantenendo le caratteristiche desiderate, come `Stream<T> limit(long maxSize)` che ritorna uno Stream troncato ad una lunghezza non superiore a `maxSize`.

Split Iterator

Per esprimere una sorgente che supporti parallelizzazione non sono sufficienti `Supplier` e `Iterator`, infatti servono metodi che consentano alla sorgente di esplicitare le opportunità di

suddivisione dello stream in rami di esecuzione indipendenti.

```
public interface Spliterator<T>
```

Il metodo di avanzamento è `tryAdvance()` non è l'utilizzatore che ottiene il nuovo elemento, ma è lo stream che fornisce l'elemento al codice che deve operarci sopra

```
// Se rimangono elementi prova a eseguire l'azione, ritornando vero, altrimenti
falso.
boolean tryAdvance(Consumer<? super T> action); // action che va ad eseguire la
pipeline per il prossimo evento
```

Questa classe offre anche metodi per: `long estimateSize()` stimare il numero degli elementi rimanenti\ `int characteristics()` ritorna un set (flag dello stream) di caratteristiche della sorgente, `default void forEachRemaining(Consumer<? super T> action)` esegue un'azione sequenzialmente per ogni elemento rimanente fino a quando non terminano gli elementi o viene lanciata un'eccezione.

Operazioni terminali

```
T reduce(T identity, BinaryOperator<T> accumulator);
<R, A> R collect(Collector <? super T, A, R> collector); // fa un operazione di
riduzione sugli elementi (più efficiente)
```

La classe `Collectors`

Permette di gestire operazioni terminabili dove l'accumulatore è un oggetto mutabile (per efficienza)
Permette di produrre dei `Collector` a partire dagli elementi di base:

- `Supplier<A>` del contenitore di risultato.
- `BiConsumer<A, T>` che accumula un elemento nel contenitore.
- `BinaryOperator<A>` che combina due contenitori parziali.
- `Function<A, R>` che dal contenitore ottiene il risultato finale.

Parallelismo

Lo stream di default decide autonomamente quanto parallelismo usare, attraverso `ForkJoinPool`.

Blocking factor

Intensità del calcolo di un algoritmo (quanto l'algoritmo è incline a bloccare la CPU). Un algoritmo con BF = 0 occupa costantemente la CPU. Un algoritmo con BF = 1 è costantemente in attesa di I/O.

$$\text{vogliamo tenere il n. threads} \leq \frac{\text{n. cores}}{1 - \text{BF}}$$

Quindi se l'algoritmo effettua molta I/O può essere utile aumentare il numero di Thread disponibili.

Programmazione distribuita

Tecniche per la gestione di più processi su macchine diverse che operano in modo coordinato allo svolgimento di un unico compito.

Motivazioni e caratteristiche

- **Affidabilità**: fare in modo che le attività proseguano anche con malfunzionamento/errore di un nodo
- **Suddivisione del carico**: il lavoro può essere suddiviso tra più nodi ed essere eseguito concorrentemente.
- **Diffusione**: gli utenti possono accedere al risultato da uno qualsiasi dei nodi.

Caratteristiche di un algoritmo distribuito

- **Concorrenza dei componenti** i vari nodi di esecuzione operano contemporaneamente (la distribuzione estende la concorrenza)
- **Totale asincronia**: non si può imporre ordine temporale degli eventi che avvengono su nodi diversi (se fosse necessario sarebbe costoso)
- **Fallimenti imperscrutabili**: il fallimento di un nodo è indipendente dagli altri ed è indistinguibile da un ritardo di consegna.

Messaggi e metodi

I nodi comunicano tra loro scambiandosi messaggi (come unica risorsa condividono la rete). Uno dei primi tentativi di facilitare invio e ricezione dei messaggi in un sistema distribuito è l'**RPC** - *Remote Procedure Call*, un sistema che rende trasparente la localizzazione del codice chiamato (cerca di rendere la chiamata simile a una chiamata locale)

Passi di una RPC

1. Il client chiamo lo *stub* locale (uno *stub* è il componente che adatta questa chiamata).
2. Lo *stub* mette i parametri in un messaggio (*marshalling*).
3. Lo *stub* invia il messaggio al nodo remoto.
4. Sul nodo remoto, un *server stub* attende il messaggio ed estrae i parametri (*unmarshalling*).
5. Il *server stub* chiama la procedura locale.

Nei linguaggi Object Oriented questo meccanismo prende il nome di **RMI** - *Remote Method Invocation*, in quanto l'obiettivo è rendere lo scambio di messaggi equivalente alla chiamata del metodo in un oggetto locale. Oltre alla comunicazione, c'è il problema di come indirizzare l'oggetto destinatario della chiamata che è presente solo nella memoria remota (il server che offre questo servizio è detto ***Object Broker***).

CORBA

Common Object Request Broker Architecture standard che descrive gli oggetti tramite un linguaggio IDL (*Interface Definition Language*) che gli consente di far interagire tecnologie differenti. L'implementazione per ciascun linguaggio genera la parte di *stub*, client e server.

Passi per una chiamata RMI (Java)

1. Il client chiama lo *stub* locale.
2. Lo *stub* prepara i parametri in un messaggio e lo invia. Il client è bloccato (aspetto il risultato).
3. Il server riceve il messaggio, lo controlla e cerca l'oggetto chiamato.
4. Il server recupera i parametri e chiama il metodo destinatario.
5. Il server prepara il risultato in un messaggio e lo invia allo *stub*
6. Lo *stub* lo riceve e recupera il risultato, ritornandolo al client.

Problematiche

- Le reti possono essere inaffidabili
- Difficile per sistemi di decine o centinaia di nodi essere tutti aggiornati alla stessa versione software
- Firewall, reti temporanee e wireless rendono impossibile indirizzare liberamente un singolo terminale (è difficile che spostandosi si riesca a dargli un indirizzo che non cambia nel tempo).
- I nodi entrano ed escono facilmente da una rete e più sono più è facile che qualcuno di essi fallisca
  Quindi le tecnologie RMI sono poco utilizzate e `java.corba` è deprecato.

Serializzazione

Un elemento fondamentale della comunicazione RMI è la *serializzazione*.
Usare un meccanismo di codifica che prende l'oggetto e lo traduce in messaggio (corrisponde al passo di *marshalling*). La deserializzazione corrisponde all'*unmarshalling*. Java ha un meccanismo di serializzazione nativo, `java.io.Serializable`.

> ⓘ ***Marker interface***
>
> Interfaccia che non ha metodi ma serve solo a segnalare che quell'oggetto può essere trattato in modo particolare.

Problemi della serializzazione

- gestire il cambiamento strutturale delle classi
- serializzare gradi di oggetti (ovvero oggetti che contengono altri oggetti. Es: se ci fosse una dipendenza circolare)
- indicare che oggetti non possono / non devono essere serializzati
- affidabilità e integrità dei dati serializzati
- rendere *marshalling/unmarshalling* efficienti in questioni di tempo / spazio
  Quindi è sconsigliata, si preferiscono utilizzare protocolli testuali trasportati da HTTP (invece che protocolli binari). Serializzazione ⇒ unmarshalling, protocolli seriali ⇒ parsing (controlli di integrità e autenticità + facili).

Primitive di Networking

Esiste un unità più bassa del Thread per gestire la concorrenza, chiamata Fiber. Il principale utilizzo delle Fiber è aumentare l'efficienza nel gestire tante connessioni insieme, perché affidare una singola connessione a un thread intero non scala bene. (Es: un server gestito con thread può supportare decine di migliaia di connessioni, mentre se si basa su fiber milioni).

`Socket` (TCP)

È un'astrazione per la comunicazione bidirezionale punto-punto tra due sistemi. Un `Socket` rappresenta un collegamento attivo. Lato client (`Socket`) lo diventa appena l'handshake TCP/IP è completato. Lato server (`ServerSocket`), viene ritornato quando un collegamento è ricevuto e completato.

```
public Socket accept() throws IOException // rimane in ascolto per una connessione (bloccante)
```

Socket implementa `Closeable`, il che significa che può essere usato nel costrutto try-with-resources.

Input e output stream

Un `Socket` (client o server) fornisce un `InputStream` e un `OutputStream` per ricevere e trasmettere dati nel collegamento (basta chiamare il getter). Questi stream sono sottoposti a diverse regole:

- Sono *thread-safe* ma solo un thread può scrivere o leggere per volta (pena eccezioni).
- I buffer sono limitati e in alcuni casi i dati in eccesso possono essere scartati sileziosamente (se non si leggono i byte abbastanza velocemente o se si scrive troppo velocemente).
- Lettura e scrittura possono bloccare il thread.
- Alcune connessioni possono avere caratteristiche particolari (ad es flag urgent data per certi pacchetti).

Una volta terminata la comunicazione, il `Socket` va chiuso esplicitamente. Se ci sono dei thread bloccati in `accept()` verrà lanciata una `SocketException`. Appena il server riceve il carattere `\n`, `BufferedReader::readline` ritorna ed il server risponde.
Siccome la comunicazione avviene via `Stream`, non si può sapere se la richiesta è terminata o meno, né si è in grado di segnalarlo (questo sta all'implementazione).

`Datagram` (UDP)

Un `Datagram` è un'astrazione per l'invio di un pacchetto UDP verso **una o più destinazioni**. **Non c'è garazia di ricezione o ordinamento in arrivo**.

```
public final class DatagramPacket
```

La dimensione massima è di 64KB (oltre questa soglia detta MTU viene frammentato in più pacchetti). Per client e server abbiamo bisogno di una sola classe, ovvero `DatagramSocket(int`

`port)`, dotato di metodi `send` e `receive`.
Con i `Datagram` la logica del protocollo è differente:

- La dimensione del messaggio nota
- Possibilità di *multicast*

Ma rispetto ai `Socket` perdiamo:

- *Affidabilità*: non c'è garanzia/segnale di consegna
- *Reciprocità*: una sola direzione (la risposta richiede di mettersi in ascolto).
- *Dimensione*: messaggi lunghi fortemente penalizzati nell'affidabilità

`URL`

GET

```
InputStreamReader ir = new InputStreamReader(new
URL("https://test.com").openStream());
BufferedReader br = new BufferedReader(ir);
String line = br.readLine();
```

POST

Per effettuare una `POST` invece bisogna esplicitarlo ottenendo la `connection`:

```
URL url = new URL("https://test.com/post");
URLConnection conn = url.openConnection();
conn.setDoOutput(true);

PrintWriter pw = new PrintWriter(conn.getOutputStream());
pw.println("test=val");
pw.close();
```

La classe `HttpClient`

Più versatile, maggior controllo di come vengono effettuate le richieste. La costruzione del client e delle richieste segue il *Build Pattern*: vengono concatenate a `HttpRequest.newBuilder()` chiamate a metodi che ritornano un oggetto la cui configurazione viene via via completata (uri, timeout, header, <METHOD>). Per fornire il contenuto di una `POST` o `PUT` si usa un parametro di tipo `BodyPublisher`
Una volta costruita la richiesta, la si può eseguire con `client.send(request, BodyHandlers.ofString())` per ottenere risposta. Un *builder* non ancora completato può essere usato più volte, copiato/modificato. La classe `BodyHandlers` contiene strategie comuni per gestire il corpo della risposta.

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
 .POST(BodyPublisher.ofString("foo=bar"))
 .header("Content-Type", "")
 .uri(URI.create("url"))
 .build()
```

CompletableFuture

Una richiesta può essere inviata anche in modo asincrono e si ottiene un `CompletableFuture`, che accetta istruzioni da eseguire al completamento del calcolo.

```
client.sendAsync(request, BodyHandlers.ofString()) // body handler permette di
gestire il formato della risposta
.thenApply(HttpResponse::body)
.thenAccept(System.out::println);
```

`Channel`

Astrazione per unificare le operazioni di I/O su canali differenti (file, rete, hardware...). (Implementa `Closeable`). Le varie implementazioni si occupano della gestione delle risorse (es `NetworkChannel`).

`NetworkChannel`

Rappresenta una comunicazione su rete. Può

- essere legato (con `bind`) a un indirizzo
- dichiarare le opzioni che supporta (es: multicast)

`AsynchronousServerSocketChannel`

È un canale asincrono basato su un server socket. Permette in modo asincrono di accettare connessioni e gestirle.
`CompletitionHandler<V, A>` è l'interfaccia che deve implementare un oggetto che gestisce la ricezione di un'operazione di I/O asincrona.

```
void completed(V result, A attachment); // gestire l'interazione relativa ai dati
ricevuti
void failed(Throwable exc, A attachment); // interazione fallita
```

Quindi implementando un `CompletitionHandler` possiamo esprimere il comportamento del server alla prossima interazione, in modo asincrono.
Il parametro generico `attachment` permette di far circolare le informazioni di contesto riguardo allo **stato della conversazione**.
I vari handler potrebbero essere chiamati da `Thread` diversi, in momenti imprevedibili, quindi per questo motivo è necessario specificare sempre l'attachment da far circolare ed il `CompletitionHandler` che gestisce il suo completamento. In alternativa all'uso del `CompletitionHandler` si possono utilizzare i metodi che ritornano un `Future`.

> **Questo richiede di riorganizzare il codice ma permette di gestire molte più connessioni**

Fallacies and frameworks

8 fallacies of distributed computing

> ⓘ **Fallacia: Che può trarre in inganno, perché sembra vero**

1. ***The network is reliable***: non solo un guasto hardware, per molti motivi una rete può essere inaffidabile. Un software distribuito deve tenere conto della possibilità che la connessione si interrompa, venga negata o che i messaggi vadano persi. Necessità di bilanciare il costo della ridondanza e dell'affidabilità.
2. ***Latency is zero***: c'è limite fisico alla velocità di un messaggio (la velocità della luce) e bisogna tenerne conto se il tempo è importante.
3. ***Bandwith is infinite***: nonostante la banda a disposizione stia aumentando, anche la quantità di dati aumenta. Se è particolarmente grande, va gestito come insieme di parti trasferite singolarmente (in genere si cerca di dare priorità all'affidabilità piuttosto che all'economia di trasmissione).
4. ***The network is secure***: se il contenuto del messaggio ha informazioni confidenziali, il protocollo deve essere reso sicuro (autenticato, autorizzato, integro e confidenziale)
5. ***Topology doesn't change***: i dispositivi sono in movimento, quindi la topologia è in continua evoluzione (si deve utilizzare un indirizzamento dinamico, che non dipende dalle caratteristiche fisiche della rete).
6. ***There is one administrator***: sempre più raramente si riesce a imporre requisiti al percorso delle connessioni perché sempre più organizzazioni sono coinvolte nella loro gestione (e requisiti particolari potrebbero non essere soddisfatti da uno qualsiasi dei nodi).
7. ***Transport cost is Ø***: sta aumentando l'importanza del bilancio energetico (uno dei costi più importanti)
8. ***The network is homogeneous***: le reti non sono più costituite da un solo mezzo di comunicazione, quindi il protocollo di comunicazione deve evitare di richiedere caratteristiche di trasporto particolari.

Frameworks

Fornisce un ambiente all'interno del quale un insieme di casi d'uso fondamentali è reso facile, efficiente e sicuro da implementare. Ad esempio un framework per applicazioni web rende facile *specificare le rotte a cui rispondere* e *costruire le risposte*.\ Il framework web cercherà di rendere trasparente e configurabile:

1. gestione dei dettagli del protocollo
2. sicurezza nel trattamento della comunicazione
3. suddivisione delle risorse tra le parti del sistema

Perché non usare un framework

- Un caso d'uso non tra quelli prescelti come importanti dagli autori del framework può essere complesso da implementare. Gli utenti dipendono completamente dalle scelte degli autori, che danno maggiore priorità a semplicità d'uso, sicurezza o efficienza.

- Potrebbe fallire l'esecuzione di una richiesta a causa di un bug nell'implementazione del framework
- Falle di sicurezza a causa di default errati
- Potrebbe permettere il sovraccarico del sistema a causa di una mancata limitazione di risorse

Stato distribuito

Le stesse caratteristiche di un sistema distribuito possono essere desiderabili anche per un insieme di dati:

- sempre disponibili anche con guasti
- quantità superiore a quella gestibile da una sola macchina
- accessibilità da più posizioni geografiche
  Un sistema di dati gestito da un sistema distribuito è accessibile e coerente **solo quando i singoli nodi sono allineati e coordinati tra loro**.

DNS

La mappatura host name a indirizzi IP era mantenuta in un file `hosts.txt`, che era condiviso in FTP a tutti gli host. Tuttavia la crescita esponenziale delle reti e la necessità delle organizzazioni locali di amministrare i propri nomi e indirizzi, hanno portato alla nascita di un database distribuito. Nel *Domain Name System* ogni nodo è responsabile di un ramo di un albero degli indirizzi. Un client chiede la risoluzione di un nome al DNS più vicino, il quale eventualmente propaga la richiesta.

Problema del consenso

Siccome la resistenza ai guasti è particolarmente importante, è necessario replicare lo stato su più di un nodo, in modo che:

- un guasto non porti perdite di dati
- aggiungendo nodi aumenti la capacità di risposta del sistema
  Avendo più nodi che contengono lo stesso dato, nasce il *problema del consenso*, ovvero di garantire che tutti i nodi contengano la stessa versione di un dato. Questo è stato denominato problema dei ***Generali Bizantini***

> ❝ Problema dei generali bizantini
>
> Diversi generali di un esercito Bizantino assediano una città. Devono raggiungere un consenso unanime su una decisione: attaccare o ritirarsi. Un attacco parziale è svantaggioso rispetto a un attacco coordinato o a una ritirata completa. Possono comunicare solo scambiandosi messaggi l'uno con l'altro. Tuttavia, non si sono garanzie sulla loro lealtà: alcuni generali possono tradire, inviando messaggi contraddittori o non rispondendo ai messaggi ricevuti.

Il problema modella il caso in cui un nodo di un sistema distribuito si comporta in modo diverso ad ogni risposta. Di fatto, non è conoscibile dall'esterno se il suo comportamento è corretto o se il

nodo è guasto. Lo strumento per affrontare la soluzione di tale problema è *l'algoritmo del consenso* o ***PAXOS***.

PAXOS

È basato su un protocollo a tre fasi che garantisce l'assenza di blocchi nel caso di un guasto singolo. I possibili comportamenti dei nodi sono modellati con delle macchine a stati per garantire la copertura di tutti i casi. Nell'algoritmo PAXOS i nodi assumono dei precisi ruoli: *leader, votanti, ascoltatori, proponente, client*.
Il *proponente* inoltra la richiesta del *client* ai *votanti* per raggiungere un quorum. Una volta che un sufficiente numero di votanti appartiene a un quorum, la richiesta è accolta. Gli *ascoltatori*, ricevendo messaggi dai *votanti* forniscono ridondanza in caso di guasti (cioé possono sostituire i *votanti* nel caso in cui non rispondano nei tempi previsti).
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).
Una variante dell'algoritmo è in grado di fornire garanzie di correttezza anche in presenza di Guasti Bizantini (nodi che non cooperano).

RAFT

Algoritmo del consenso scritto con l'obiettivo di essere più comprensibile di PAXOS. Suddivide il problema del consenso in sottoproblemi per rendere il modello più semplice. Il concetto principale è la replicazione di una macchina a stati.

Cap Theorem (Compromessi)

Tramite il consenso, il sistema distribuito ha uno stato coerente (ogni nodo può rispondere allo stesso modo alla richiesta di un dato). Il teorema CAP definisce 3 caratteristiche di un database distribuito e afferma che solo due di esse possono essere contemporaneamente garantite:
C (*Consistency*) Ogni lettura riceve il valore più recente o un errore
A (*Availability*) Ogni richiesta riceve una risposta valida (non necessariamente l'ultimo valore)
P (*Partition Tolerance*) Tolleranza alla separazione: il sistema funziona anche se la rete fallisce per un insieme di nodi, cioè viene *partizionata*.
Il teorema afferma che solo due delle proprietà CAP possono essere garantite contemporaneamente, quindi:
In caso di partizione di rete un sistema può essere o consistente o disponibile, ma non entrambe le cose. Ovvero: la consistenza è garantita, ma alcune richieste vanno in errore, oppure tutte le richieste ritornano un valore, ma alcune potrebbero non tornare quello più recente.

PACELC

estensione del *Cap Theorem*, che considera anche il caso di operatività normale: in caso di (P) partizione, si deve scegliere tra:

- (A) disponibilità e (C) consistenza
  (E) altrimenti, tra
- (L) latenza e (C) consistenza.
  In genere, i db NoSQL si orientano verso PA/EL, mentre quelli relazionali PC/EC.

CRDT (Conflict-Free Replicated Data-Type)

Finora abbiamo assunto che i dati vengano scritti da un'unica fonte, tuttavia nei sistemi distribuiti è normale che ogni nodo abbia informazioni da fornire, che devono essere riunite con quelle prodotte dagli altri nodi. In questi sistemi, l'esigenza non è il *consenso*, ma riuscire a riunire i cambiamenti prodotti localmente da ogni nodo con quelli prodotti dagli altri nodi. 2 soluzioni a questo problema:

- *Operational Transformation*: ogni nodo produce delle modifiche, che vengono propagate agli altri nodi. Ogni nodo trasforma le modifiche ricevute in modo da applicarle al suo stato del documento, quindi quello che viaggia è il cambiamento non lo stato. (Usato ad es. da *Google Docs*). Non ha avuto molto successo per la sua complessità di implementazione.
- *Conflict-Free Replicated Data-Type* una CRDT può essere replicata su più nodi, modificata indipendentemente e contemporaneamente, con la garanzia che esiste un modo per risolvere tutti i conflitti. Strutture dati con queste caratteristiche: grow-only counter, positive-negative counter, grow-only set, 2-phase set, last-write-wins set.

CALM theorem

Permette di affrontare il problema del consenso distribuito in contrasto con il CAP theorem (ovvero, programmi che possono mantenere la consistenza anche in caso di partizionamento). CALM sta per *Consistency As Logic Monotony*, cioé i programmi che mantengono le proprietà CP possono essere espressi in termini di *logica monotona*. Il CALM theorem quindi individua una classe di programmi che forniscono una garanzia. Il problema rimane che questa classe di programmi è poco popolata.

> ⓘ **Sistema logicamente monotono**
>
> Un programma è *logicamente monotono* se all'aumentare della dimensione del problema il risultato non cambia. Se ho ad esempio un sistema distribuito con 10 nodi che individua un deadlock al suo interno. Anche con più nodi, la soluzione non cambia.

> ⓘ **Sistema non logicamente monotono**
>
> Esempio: Sistema distribuito di Garbage Collection. Aggiungendo nodi, potrebbe accadere che all'interno di uno di essi ci sia un riferimento ad un oggetto che era stato classificato come non utilizzato. Quindi il sistema cambia aggiungendo nodi.

Reactive extensions

Per cercare astrazioni di livello più alto per gestire le problematiche a sistemi concorrenti o distribuiti, si può utilizzare il concetto di `Stream`, il quale è basato sull'inversione del controllo dell'iterazione. Tuttavia:

- manca un protocollo esplicito per la terminazione degli stream

- gli errori sono gestiti come eccezioni

Vengono introdotte in .NET le ***Reactive Extensions***, una libreria per comporre codice asincrono e event-based utilizzando sequenze osservabili (semantica più precisa dello stream per elaborare sequenze asincrone di eventi). Le basi del modello sono i seguenti concetti: Observable, Scheduler, Subscriber, Subject

Implementazione dell'Observer Pattern *done right*

Un `Observable` in Rx è concettualmente simile ad uno stream, che emette nel tempo una sequenza di valori. Si possono osservare i valori emessi da un `Observable` fornendo il comportamento da adottare in caso di: valore ricevuto, eccezione, termine flusso di dati.

`Scheduler`

La maggior parte degli operatori sugli `Observable` accettano uno `Scheduler` come parametro, che permette di indicare il tipo di concorrenza desiderato. Es: `Schedulers.computation()` indica un'esecuzione fortemente basata sulla CPU.

`Subscriber`

Rappresenta un ascoltatore di un `Observable`: fornisce il codice per reagire agli eventi

`Subject`

Si può comportare sia da `Observable` che da `Subscriber` (rimanendo in ascolto di un `Observable`) e comportarsi esso stesso da `Observable`
Lo schema concettuale proposto da Rx è utile per:

- Costruire stream di elaborazione complessi e asincroni
- Fornire un'interfaccia semplice per trattare successioni di eventi
- Scrivere algoritmi facili da portare da un linguaggio ad un altro
- Strutturare un'elaborazione concorrente di uno stream di dati su garanzie solide
- Gestire con semplicità gli eventi provenienti dalla UI (come fossero dati provenienti da file o da altre sequenze di dati).

Reactive Manifesto

Definisce le caratteristiche dei sistemi reattivi:

1. *Responsive* (pronti alla risposta): in quanto la bassa latenza di interazione è fondamentale per l'usabilità. È importante fornire una risposta in tempi prevedibili e costanti (anche se è un errore, è sempre una risposta).
2. *Resilient* (resilienti): gestire i fallimenti continuando a rispondere con la stessa prontezza (replicazione di componenti isolati, coscienza che ogni parte del sistema può fallire, creazione di componenti sostitutivi a quelli andati in errore).
3. *Elastic* (elastici): in grado di consumare una quantità variabile di risorse in funzione del carico in ingresso. Si cerca di distribuire il carico in modo da evitare bottleneck o punti di conflitto, suddividendo gli input in *shard*

4. *Message Driven* (orientati al messaggio): questa primitiva di comunicazione abilita le altre caratteristiche. Attraverso lo scambio di messaggi i componenti possono rimanere disaccoppiati. È il mezzo con cui il sistema si struttura nella forma *elastica* e *resiliente* per ottenere la *prontezza nella risposta*.

Big vs Fast

Fast Data

Dati di dimensioni paragonabili ai Big Data, in arrivo continuo. È impensabile elaborarli e/o persisterli tutti, quindi è necessario trattarli via via che si presentano.\ Nell'estrarre dati da un flusso, il principale problema è **armonizzare** le differenti **velocità di elaborazione** tra i vari componenti. L'esecuzione di *batch* paralleli non poteva più funzionare in quanto introduceva latenze superiori al periodo di utilità delle info estratte (il componente più lento stabiliva la velocità massima di elaborazione). Da questo problema nasce la necessità di aggiungere alle Reactive Extensions il concetto di *back-pressure*.

> **Back-pressure**
>
> Resistenza che il componente successivo può opporre a dati provenienti dal componente precedente nella catena di elaborazione (concetto che permette ad ogni nodo della catena di dichiarare quanti dati è in grado di gestire).

Reactive Streams

Aumenta le garanzie fornite da ReactiveX introducendo l'esplicita gestione della *back-pressure* per impedire che un nodo possa essere soverchiato dai dati inviati dal nodo precedente, o possa soverchiare quello successivo.
Inoltre, viene inclusa anche la casistica in cui i diversi componenti di uno Stream non si trovino sullo stesso nodo, ma siano distribuiti.

Technology Compatibility Kit (TCK)

Utilizzato per verificare un'implementazione candidata. Implementazioni che lo soddisfano: MongoDB java Driver, RxJava, Vertx Reactive
Streams...
Il modello attuale di Reactive Streams è compatto:

- `Publisher` (Observable): fornisce un numero potenzialmente infinito di elementi in sequenza
- `Subscriber`: consuma gli elementi forniti da un `Publisher` ed è in grado di controllare il flusso degli elementi in arrivo
- `Subscription`: rappresenta il legame tra `Publisher` e `Subscriber` e permette di controllarlo, ad esempio interrompendolo
- `Processor` (Subject): sia un `Subscriber` che un `Publisher` e deve sottostare ad entrambi i contratti

Operatori

`map`, `flatMap` (produce un numero arbitrario di output per ogni input, quindi `0` o più), `filter`, `skip` (emette uno stream saltando i primi n elementi della sorgente), `zip` (emette uno stream combinando a coppie elementi di due stream in ingresso), `debounce` (emette un elemento solo se è passato un lasso di tempo dall'ultimo elemento della sorgente), `window` (divide lo stream sorgente in gruppi, ad es: per tempo, per caratteristica, per n. di elementi...).

Schedulatore

Le caratteristiche dell'asincronia possono essere diverse a seconda dello Schedulatore usato

- `.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

In `RxJava` l'operatore `parallel()` permette di indicare che uno stream va costruito come parallelo. In questa modalità, non tutti gli operatori sono consentiti e va specificato lo schedulatore da usare con il metodo `onRun(scheduler)`. Il metodo `sequential()` indica che da quel punto in poi la pipeline va messa in sequenza.

Attori

`RxJava` non è conforme al *Reactive Manifesto* siccome non è basata sullo scambio di messaggi. Il **modello ad attori** è in grado di soddisfare il modello *Reactive* e i requisiti del *Reactive Manifesto*.

Caratteristiche

Ogni attore è un'unità indipendente di elaborazione, con uno stato privato inaccessibile dall'esterno. L'attore comunica solo con messaggi diretti ad altri attori di cui conosce il nome. In un determinato momento, un attore ha un **comportamento** che definisce la sua reazione ad un determinato messaggio. Per reagire a un messaggio un attore può:

- mutare il proprio stato
- creare nuovi attori
- inviare messaggi ed attori noti
- cambiare il suo comportamento

Concorrenza

All'interno dell'attore non c'è concorrenza, l'esecuzione è sequenziale. Al di fuori è tutto concorrente e distibuito:

- un altro attore può trovarsi dovunque
- ogni attore è concorrente agli altri
- ogni messaggio è concorrente agli altri
  Se un attore fallisce, non si ha alcuna conseguenza sulla stabilità del programma (il fallimento

è considerato normale). Un attore può supervisionare quelli che ha creato ed essere notificato del suo fallimento. In questo sistema:

- la creazione di nuovi attori è economica
- alta affidabilità, efficienza e scalabilità
- distribuzione e concorrenza sono caratteristiche primarie
  Permette l'elaborazione di grandi moli di dati asincroni, spesso modellati come eventi.

Problematiche del modello

- Modellare un algoritmo come inerazione di un insieme di attori concorrenti non è sempre naturale
- Tecniche di programmazione tradizionale sono inutili/dannose per questo paradigma
- L'interazione non può fare ipotesi né sull'ordine della ricezione dei messaggi, né sulla loro affidabilità.
  Il **Framework Akka** è la più diffusa implementazione del modello ad attori sulla JVM (scritto in Scala).

Behaviour

Il comportamento di un attore è una funzione dal tipo di messaggio ad un effetto (a sua volta una funzione che trasforma il vecchio comportamento nel nuovo). Quindi è il compilatore che impedisce di inviare messaggi che non possono essere ricevuti.

Produzione di attori

Con il metodo `actorOf` che è parametrico sul tipo di messaggio e il comportamento iniziale.

Server(less)

I programmi Java vengono tradotti in bytecode ed eseguiti dalla JVM, la quale applica strategie per bilanciare l'efficienza di esecuzione (il che ha un costo in termini di tempo di avvio del programma). Quando Java e la JVM sono stati inventati, la performance aveva livelli da accettabili a ottimi. Al giorno d'oggi è penalizzata.

> ⓘ **Servizi *serverless***
>
> Un servizio cloud *serverless* permette di acquisire le risorse necessarie all'esecuzione in una sola richiesta, con il solo costo del tempo per costruire la risposta. Un server non è più sempre in attesa delle richieste, esiste solo per il tempo di erogare la risposta.
>
> Il tempo di startup è uno svantaggio per la JVM.

GraalVM

Progetto di riscrittura del compilatore Java in Java stesso. Include:

- un nuovo compilatore JIT (just in time)

- un compilatore ahead-of-time per Java (questo limita il codice java perché alcune feature sono dinamiche)
- una specifica di codice intermedio (*Truffle*)
  GraalVM è in grado di compilare un qualsiasi bytecode da un linguaggio sulla JVM in un eseguibile nativo (indipendente da JVM, il risultato è binario). Attraverso Truffle può supportare anche altri linguaggi.
  Si ottiene una sostenuta riduzione dell'eseguibile finale, che comprende anche parti necessarie della libreria standard, ma soprattutto del tempo di avvio.

Java to Container - Micronaut

Framework che supporta GraalVM, si può ottenere un eseguibile nativo già in un container.