Cortosi Vittorio

Programmazione 3

- **OOP**
  - **Alan Key**
  - **Principi fondamentali**
- **Ereditarietà**
  - **TypeCasting**
- **Visibilità dei membri**
- **TypeChecking**
- **Classi Astratte**
- **Interfacce**
- **Java Reflection**
- **Java Generics**
  - **Wildcards in Java Generics**
- **Nested Class**
- **Streams**
  - **Gerarchia**
  - **Bufferizzare gli stream**
  - **Serializable interface**
- **Programmazione ad eventi**
- **Observer Pattern**
- **PatternMVC**
- **JavaFX**
  - **EventHandlers**
  - **CSS in JavaFX**
  - **JavaFX con java Collection**
  - **JavaFXML**
  - **JavaFXML – Gestione degli eventi**
  - **JavaFX – JavaBeans e Properties**
- **Threads**
  - **Thread Daemons**
- **Sincronizzazione**
  - **Metodi sincronizzati**
  - **Lock intrinsechi**
  - **Blocchi sincronizzati**
  - **Blocchi custoditi**
- **Java Pipes**
- **Blocking queue interface**
- **Lock Interface e Condition**
- **ReadWriteLock**
- **ThreadPools**
- **Callable Interface e Future Interface**
- **Java Socket**

OOP

Object Oriented Programming è un paradigma di programmazione che considera il programma costituito da oggetti (o istanze) che possono agire da soli o interagire tra loro.

Nel programma, gli oggetti software possono essere entità scollegate dalla realtà oppure correlate/rappresentano un oggetto del mondo reale (o una sua astrazione).

Ogni oggetto sistema/scenario è dotato di:

- Identità: è riconoscibile all'interno del sistema
- Diverse caratteristiche (attributi)
- Stato: I valori degli attributi di un oggetto costituiscono lo stato dell'oggetto
- Un comportamento: agisce a stimoli esterni (chiamate a metodi)

Gli oggetti dello stesso tipo condividono lo stesso tipo di dato (classe).

La classe di un oggetto definisce il tipo di dato dell'oggetto (la sua rappresentazione grezza senza stato). Si può immaginare la classe come una stampante di automobili: la classe sa com'è fatta l'automobile che deve stampare (ha una forma, un colore, una targa… ma senza stato), conosce anche il suo comportamento (può accelerare, può sterzare, etc.). Un'istanza di quella classe è un'automobile ma compresa di stato; quindi, sarà un'automobile con un numero di targa ben definito, un colore ben definito, etc.

La programmazione orientata agli oggetti secondo Alan Key

Alan Key, descrivendo il suo linguaggio di programmazione Smalltalk ha dato una sua interpretazione di cosa è la OOP, disse principalmente che:

- Ogni cosa è un oggetto
- Un programma è un insieme di oggetti che appartengono ad un tipo specifico (classe) e che comunicano tra loro invocando i metodi che questi oggetti offrono.
- Un oggetto può contenere riferimenti ad altri oggetti (è possibile creare nuovi tipo di oggetti a partire da quelli già esistenti. In questo modo è possibile scrivere programmi complessi nascondendone la complessità nella semplicità dei singoli oggetti).
- Tutti gli oggetti di un determinato tipo (appartenenti quindi ad una stessa classe) possono rispondere agli stessi messaggi

Principi fondamentali dell'OOP

1. **Incapsulamento e Information Hiding**. Incapsulamento è il principio secondo il cui si avvolgono/incapsulano i dati e i metodi di un'entità all'interno di una classe. L'information Hiding si occupa di proteggere i membri di una classe da un accesso illegale o non autorizzato. La differenza sostanziale è che l'incapsulation si occupa di nascondere la complessità di un programma e la sua leggibilità, l'information hiding si occupa della sicurezza dei dati.
2. **Astrazione**. Nella OOP il concetto di astrazione definisce il modo con in quale si denotano solo le caratteristiche essenziali di un oggetto nascondendone i dettagli non rilevanti. Ad esempio, quando si guida l'automobile ci preoccupiamo di utilizzare i metodi che l'automobile ci offre (accelera, frena, metti sotto il ciclista, etc.) ma non ci preoccupiamo di come siano implementate queste funzioni. Ciò è anche dovuto grazie all'incapsulamento.
3. **Modularità**. Suddividere un sistema in una serie di componenti indipendenti che interagiscono tra di loro per ottenere il risultato desiderato

Ereditarietà

Uno dei concetti chiavi dell'OOP e consiste in una relazione tra due classi: La superclasse definisce un concetto generale, mentre la sottoclasse rappresenta una variante specifica di tale concetto. **In Java l'ereditarietà è singola**.

Polimorfismo: È un concetto che descrive che qualcosa si presenta sotto molteplici forme. Il polimorfismo aggiunge flessibilità al codice che lo rende più estensibile e mantenibile.

In java il polimorfismo è applicabile sotto diversi aspetti, ad esempio il riferimento di una variabile si dice che può assumere più forme, perché essa può puntare a tipi diversi di oggetti e quindi assumere più forme.

In Java il polimorfismo viene associato alla capacità del compilatore (statico) o dell'interprete (dinamico) di riconoscere quale metodo dovrà essere associato a quella chiamata di funzione in base all'oggetto puntato dal riferimento. In fase di compilazione si presenta il Polimorfismo statico (Compile-time Polymorphism): in ogni classe è possibile definire più metodi con lo stesso nome, ma avente firma diversa. In fase di compilazione è possibile definire quale metodo dovrà essere chiamato in base al tipo/numero/ordine dei parametri. In fase di Runtime invece entra in gioco il Polimorfismo Dinamico (Dynamic Binding o Late Binding): avendo implementato un meccanismo di ereditarietà, la stessa interfaccia (i metodi della classe padre) può essere re-implementata anche nelle sottoclassi (@Override). Questo fa sì che esistano stessi metodi avente implementazione/comportamento differente. Quindi, in fase di Runtime, si determina quale metodo dovrà essere invocato ad ogni chiamata di un metodo in una gerarchia di classi in cui è presente l'overriding dei metodi.

TypeCasting

È il processo per la conversione dei tipi. In java, in un contesto di ereditarietà, esistono due tipi di TypeCasting: Parent to Child (**Downcast**) e Child to Parent (**Upcast**). **Quando si parla di conversione di tipi in java, si parla del tipo del riferimento all'oggetto e non all'istanza dell'oggetto**, l'oggetto rimane di tipo immutato.

Utilizzando l'Upcasting:

- È possibile accedere alle variabili ed ai metodi della classe genitore, i metodi definiti esclusivamente nella classe figlio non sono accessibili.
- Eseguire il cast in maniera esplicita ed implicita:
  - Implicita: `Parent p = new Child();`
  - Esplicita: `Parent p = (Parent) c;`

Utilizzando il Downcasting:

- È possibile accedere a tutti i metodi e le variabili di entrambe le classi
- Eseguire solo il Cast in maniera esplicita:
  - Esplicita: `Child c = (Child) p;`
  - Non è consentito l'implicita: `Child c = new Parent();`

Visibilità dei membri

Java ha quattro livelli di visibilità che possono essere associati ai membri di una classe:

- Default: Visibile all'interno del package
- Public: Visibile ovunque

- Protected: Visibile nelle sottoclassi e nello stesso package
- Private: Solo all'interno della classe stessa

Java member visibility rules across packages:

	Member Visibility Rules						
	Access from			Access to			
	Package	Class/ Interface?		public	*package*	protected	private
1	same	same		✓	✓	✓	✓
2		inner		✓	✓	✓	✓
3		subclass		✓	✓	✓	
4		other		✓	✓	✓	
5	other	subclass		✓		inheritance	
6		other		✓			

Type Checking

Il meccanismo secondo il quale viene controllato che i valori assegnati alle variabili siano dei tipi ammissibili per le variabili. Questo meccanismo può avvenire sia in fase di compilazione (Type Checking Statico) sia in fase di esecuzione (Type Checking Dinamico). Questi controlli servono a garantire che il programma sia TypeSafe (La probabilità di errori di tipo è ridotta al minimo).

Esempio di violazione di semantica:

```
int x = 2 + "a";
```

Classi Astratte

Java implementa l'astrazione utilizzando la keyword "abstract".

Vengono utilizzate per fornire alcune funzionalità comuni a un insieme di classi correlate. Lo scopo di una classe astratta è fungere da concetto base per le sottoclassi. Una classe astratta non può essere istanziata, può essere anche assente di metodi abstract (però se si definisce un metodo abstract, la classe deve essere abstract). I metodi abstract non devono contenere implementazione (graffe).

Al suo interno è possibile quindi definire metodi (senza corpo) e attributi con qualsiasi tipo di visibilità. Può anche esserci un costruttore ma non è comunque possibile istanziare oggetti di classi astratte.

Si utilizzano nel caso in cui:

- Quando si vuole condividere codice con diverse classi strettamente correlate tra loro
- Le classi che dovranno estendere la classe astratta abbiano molti attributi in comune.
- Si vogliono definire campi **non static e non final** in modo da definire metodi che permettano di modificarne lo stato.

nella tua classe/interfaccia astratta, (per i membri abstact) puoi vedere solo le firme, che dovrebbero dirti cosa dovrebbe accadere ma non come, il come (l'implementazione) è astratto. l'implementazione è in una classe diversa, in un file diverso, in un progetto diverso forse anche. Non è "nascosto" come in "nascosto allo sviluppatore" ma è "nascosto" dal punto di vista del codice in uso. Un pezzo di codice che utilizza un'istanza di un'interfaccia conosce solo le astrazioni, ma l'implementazione è nascosta, il codice non saprà quale implementazione verrà utilizzata

Interfacce

È un modo per ottenere l'astrazione in java come per le classi astratte e quindi è una classe utilizzata come interfaccia per creare oggetti conformi al suo "protocollo". A differenza delle classi astratte, tutti i campi sono automaticamente pubblici, statici e final e inoltre è possibile implementare più di una interfaccia (una sorta di ereditarietà multipla) mentre è possibile estendere solo una classe (che sia astratta o meno).

Si utilizzano nel caso in cui:

- Quando vuoi che classi non correlate tra loro utilizzino la tua interfaccia (Tipo Comparable).
- Di specificare il comportamento di un particolare tipo di dati, senza preoccuparsi di chi implementa il suo comportamento.
- Si desidera sfruttare l'ereditarietà multipla del tipo.

Java Reflection

È una funzionalità del linguaggio che consente al programmatore di esaminare in fase di runtime gli oggetti, al fine di scoprire a che classe appartiene, la lista dei metodi, cosa implementa, cosa estende, etc.

Questo è implementato grazie al RunTime Type Identification (RTTI), che è il meccanismo che permette, in fase di Runtime, di identificare/ispezionare un oggetto in fase di Runtime.

Java utilizza l'RTTI grazie ad una serie di componenti, come il ClassLoader e l'oggetto Class. La classe Class è una classe parametrica il cui parametro T denota la classe di appartenenza dell'oggetto in questione. Per ogni oggetto definito in java viene caricato in memoria il riferimento all'oggetto Class a cui fa riferimento (oggetti dello stesso tipo fanno riferimento allo stesso riferimento).

L'oggetto class viene caricato dal ClassLoader quando si verificano uno di questi scenari:

- Viene creata un'istanza della classe con new()
- Utilizzando il metodo statico di Class: Class.forName("<ClassName>")
    - Se <ClassName> non esiste: ClassNotFoundException
- Viene invocato uno dei metodi statici della classe
- Viene assegnato un campo statico alla classe
- Viene utilizzato un campo statico di una classe che non è una variabile costante.
- Se Class è una classe di primo livello e viene eseguita un'istruzione assert annidata lessicalmente all'interno della classe.

Oltre alla classe Class, esistono molte classi all'interno di java.lang.reflect per reperire informazioni su un'oggetto, come i metodi di Field, Method, Constructor.

La reflection viene usata in tanti ambiti, come in JavaBeans.

Java Generics

Sono stati progettati per estendere il sistema di tipi di Java per consentire a classi o metodi di operare su oggetti di vario tipo fornendo al contempo la sicurezza del tipo in fase di compilazione

Secondo le specifiche del linguaggio:

- Una classe è considerata generica se implementa una o più variabili di tipo (o parametri di tipo)
- La variabile di tipo può essere qualsiasi tipo di dato che non sia primitivo.
- Un metodo è generico se dichiara una o più variabili di tipo. Queste variabili di tipo sono note come parametri di tipo formale del metodo. La forma dell'elenco di parametri di tipo formale è identica a un elenco di parametri di tipo di una classe o di un'interfaccia.
- Un costruttore può essere dichiarato generico, indipendentemente dal fatto che la classe in cui è dichiarato il costruttore sia essa stessa generica. Un costruttore è generico se dichiara una o più variabili di tipo.

I generici sono stati aggiunti a Java per garantire la sicurezza dei tipi. Per garantire che i generici non causino un sovraccarico in fase di runtime, il compilatore applica un processo chiamato **erasure** del tipo sui generici in fase di compilazione: **il parametro di tipo viene sostituito in fase di compilazione con il primo limite superiore** (se dichiarato) **o con Object altrimenti**, inoltre i parametri sulla dichiarazione di classe vengono rimossi.

I generici hanno dei militi:

- Impossibile creare un'istanza di tipi generici con tipi primitivi

```
ArrayList<int>  // No!!!!!!
```

- Impossibile creare istanze di parametri di tipo

```
T elem = new T() // Jesus no
```

- Impossibile dichiarare campi statici i cui tipi sono parametri di tipo

```
Private static T x;
```

  Il campo statico di una classe è una variabile a livello di classe condivisa da tutti gli oggetti non statici della classe. Se fosse possibile creare variabili statiche generiche, questo codice:

```
public class MobileDevide<T>

    private static T x;

    // …

}

MobileDevice<Smartphone> telefono = new MobileDevice<>();

MobileDevice<Tablet> tabler = new MobileDevice<>();
```

  Avrebbe senso sintatticamente, **ma di che tipo sarebbe x?** SmartPhone? Tablet?

- Impossibile utilizzare cast o istanze con tipi parametrizzati

- Impossibile creare array di tipi parametrizzati
- Impossibile creare, catturare o lanciare oggetti di tipi parametrizzati
- Impossibile eseguire l'overloading di un metodo in cui i tipi di parametri formali di ciascun overload vengono cancellati nello stesso tipo grezzo

```
Public class Example {

    Public void print(Set<String> strSet){}

    Public void print(Set<Integer> intSet){}
```

Non è inoltre possibile creare delle relazioni di tipo e sottotitpo con i generici: A differenza degli oggetti che non lavorano su generici

```
Object someObject = new Object();

Intero someInteger = new Integer(10);

someObject = someInteger; // OK
```

Questo codice è valido, è possibile assegnare un riferimento di Object ad un suo sottotipo.

Con i generici non è possibile (a meno che non si usino le wildcard). La seguente lista:

```
List <Integer> l1 = new ArrayList<Integer>();

List <Number> l2 = l1 // Non è consentito
```

List<Number> l1, non ha nessuna relazione con: List<Integer> l1 (Sebbene Integer estenda Number). Bisogna guardare List<Type> come un blocco solo (List<Type1> è un tipo ed List<Type2> è un altro). Quindi questo non è consentito.

Per poter creare relazioni tra tipo e sottotipo con i generici è necessario usare le **Wildcard**.

È possibile che a volte si voglia limitare i tipi che si possono utilizzare come argomenti di tipo, con dei tipi parametrici (Ad esempio un metodo che lavora su numeri, potrebbe accettare solo istanze che derivano da Number), a questo servono i tipi parametrici limitati.

Per il metodo prima citato, si scriverebbe in questo modo: <T extends Number>. Nel caso di tipi parametrici limitati, come detto all'inizio (il parametro di tipo viene sostituito in fase di compilazione con il primo limite superiore), questo non verrà sostituito con <Object> ma con <Number>.

I parametri di tipo limitato/vincolato sono fondamentali per l'implementazione di algoritmi generici. Se si volesse creare un metodo statico di libreria che conta il numero di elementi che sono maggiori di un elemento dato, avremmo:

```
public static <T> int countGreaterThan(T[] anArray, T elem) {

    int count = 0;

    for (T e : anArray)
```

```
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

L'errore di compilazione è dovuto dal fatto che l'operatore '**>**' può essere utilizzato solo con i tipi primitivi, ma non possiamo sapere con che valori lavorerà il metodo (se lavora con qualsiasi altro oggetto che non sia un tipo primitivo, avremmo bisogno di **obj.comapareTo(obj1)**, ma anche in quel caso, se lavorassimo con oggetti che non implementano Comparable? Per risolvere il problema, utilizzare un parametro di tipo delimitato dall'interfaccia Comparable<T>

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray,
T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Un parametro di tipo può avere più limiti: <T extends B1 & B2 & B3>

Se uno dei limiti è una classe, deve essere specificato per primo. Per esempio:

Classe A { /* ... */ }

interfaccia B { /* ... */ }

interfaccia C { /* ... */ }

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

Wildcards in Java generics

Il carattere jolly (?) in Java è un tipo speciale che controlla la sicurezza del tipo durante l'utilizzo dei generici (parametrizzati). Un carattere jolly viene utilizzato nei generici per rappresentare un tipo sconosciuto.

Può essere utilizzato nelle dichiarazioni e nelle istanze di variabili, nonché nelle definizioni dei metodi, ma non nella definizione di un tipo generico.

La wildcard viene usata in contesto di generici (Codice generico), **per questo motivo, non è possibile utilizzarla in contesti in cui non si lavora con codice generico** (o oggetti che fanno uso dei generici):

- Creare un'istanza di tipo sconosciuto:

```
? a; // Non è codice generico (List<?> a; lo è)
```

- Creare un'istanza di classe di tipo sconosciuto:

```
public static ? a;
```

- Dichiarare un metodo con parametro sconosciuto:

```
public method(? parameter)
```

- Dichiarare il ritorno di un metodo con parametro sconosciuto:

```
public ? method()
```

Unbounded wildcard: Per definire codice generico che operi su qualsiasi tipo di dato (ad esempio una Lista) basta usare come parametro di tipo (?).

Esempio: Un metodo che stampi tutti gli elementi di una lista (di qualsiasi tipo):

```
public static void printList(List<?> list) {
    for (Object elem : list)
        System.out.println(elem);
    }
}
```

La lista accetta qualsiasi tipo che estenda Object, è come scrivere:

```
List<? extends Object>
```

Upper bounded Wildcard: Per limitare il campo di azione su una variabile, è possibile (come nei generici) definire un limite superiore:

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

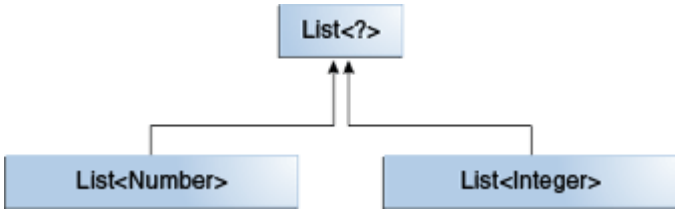
Questo metodo farà la somma degli elementi di list, solo per quelle liste che operano su Number o chi la estende.

Con l'utilizzo della wildcard è possibile creare delle relazioni di tipo e sottotipo:

Con i generics non è possibile fare questo tipo di assegnamento:

```
List <Integer> iList = new ArrayList<Integer>();

List <Number> nList = iList // Non è consentito
```

Ma con le wildcard è possibile: **il tipo comune a List<Integer> ed List<Number> è List<?>:**




```
List<Integer> intList = new ArrayList<>();

List<? extends Number>  numList = intList;  // Consentito
```

Attenzione: è possibile solo usare i metodi dell'Upperbound:

```
numList.get(0).soloMetodiDiNumber();
```

Se invece si assegna senza limitazione:

```
List<Integer> intList = new ArrayList<>();

List<?>  numList = intList;

// Sarà possibile usare solo i metodi di Object

// È come se fosse List<? extends Object>)
```

Nested Class

In java è possibile definire una classe all'interno di un'altra classe, tale classe è chiamata Classe annidata. Si possono suddividere in due categorie: Le classi annidate statiche vengono chiamate **static nested classes**, mentre le classi annidate non statiche vengono **chiamate Inner classes**

Per le static nested classes: **Non è possibile accedere ai membri della classe che la racchiude** (outer class)

Per le inner classes: È possibile accedere ai membri dell'outer class

I motivi per cui si utilizzano:

- **È un modo per raggruppare logicamente due classi che sono strettamente collegate**: se una classe è utile solo a un'altra classe, allora è logico incorporarla in quella classe e tenerle insieme.

- **Rinforza l'incapsulation e l'information hiding:** Ad esempio se avessimo due classi A e B, e B ha bisogno dell'accesso ai membri di A (che sono privati). Nascondendo la classe B all'interno della classe A, i membri di A possono essere dichiarati privati e B può accedervi. Inoltre, B stesso può essere nascosto dal mondo esterno.
- **Codice più leggibile e gestibile**: l'annidamento di piccole classi all'interno di classi di primo livello (o top-level class: classi non innestate) posiziona il codice più vicino a dove viene utilizzato.

È possibile definire una classe all'interno del corpo di un metodo, all'interno di un for o un if. Questo tipo di classe viene chiamata Local Class (classe locale).

Riepilogo:

```
public class Outer {                                    // Top-Level Class

   public class Inner1 {                    // Inner Class

   }


   public static class Inner2 {             // Static nested class

   }


   public void method(){                    // Instance method

      class LocalInner{                     // Local Class

      }

   }

}
```

Si può ulteriormente sintetizzare il codice, creando ed istanziando una classe allo stesso tempo. Vengono chiamate classi anonime (sono classi locali ma senza nome in sostanza).

Mentre le classi locali sono dichiarazioni di classe, le classi anonime sono espressioni, il che significa che vengono definire in un'altra espressione

Quando si istanzia una classe anonima, la sua espressione viene costruita in questo modo:

- Un operatore new
- Nome dell'interfaccia o classe da estendere
- Parentesi dei parametri del costruttore (con interfaccia non ci sono parametri)
- Corpo della classe anonima

Esempio: quando si clicca su un bottone vogliamo che ci stampi “Hello World”.

```
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

Il codice è più conciso.

Con le classi locali sarebbe stato:

```
class MyActionEvent implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
}
EventHandler eh = new MyActionEvent();
....
btn.setOnAction(eh);
```

Anche se l’implementazione è semplice, **se l’interfaccia o classe che stiamo implementando contiene solo un metodo, le classi anonime possono apparire alla fine ingombranti**.

Un’ulteriore semplificazione che fornisce java è quello delle **Lambda Expression**.

Questo concetto semplifica ulteriormente il codice sfruttando il fatto che esistono **interfacce con un solo metodo da implementare**. Avendo solo un metodo da implementare:

```
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
```

```
        System.out.println("Hello World!");
    }
});
```

Quello che è evidenziato in blu risulta essere un eccesso di scrittura da parte del programmatore. **EventHandler<ActionEvent>,** implementa solo un metodo: **handle()**. Sapendo questo, è possibile passare come parametro al metodo setOnAction, non una classe anonima ma bensì una sua semplificazione: espressione lambda:

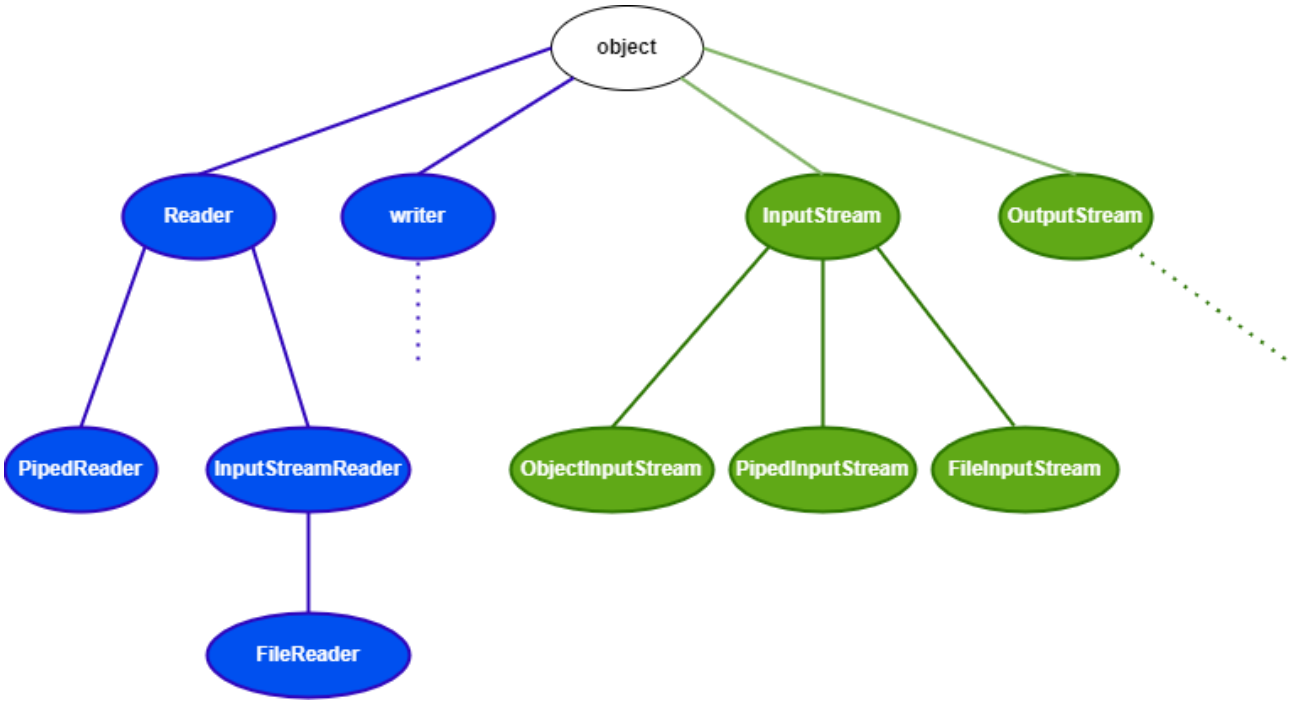
```
btn.setOnAction(event -> {
    System.out.println("Hello World!");
});
```

Streams

Per Stream I/O (input/output) ci si riferisce al trasferimento di dati da o verso un punto di memorizzazione. Uno stream può rappresentare diversi tipi di origini e destinazioni, inclusi file su disco, dispositivi, altri programmi e array di memoria.

Gli stream supportano (possono inviare e ricevere) molti tipi diversi di dati, inclusi byte semplici, tipi di dati primitivi, caratteri localizzati e oggetti. Alcuni stream trasmettono semplicemente i dati; altri manipolano e trasformano i dati in modi utili.

Gerarchia



Ciò che differenzia i due percorsi è che:

- Da Reader e Writer si specializzano tutte le classi che lavoreranno su stream di caratteri, flussi di caratteri.
- Da Input/outputStream si specializzano tutte le classi che lavorano su flussi di byte e non sono caratteri. In sostanza si usa per tutti quei file/flussi che non sono testuali.

Ad esempio, FileInputStream è usato per leggere flussi di byte mentre FileReader legge flussi di char.

Gerarchia più completa oppure Oracle

Bufferizzare lo stream

In generale si preferisce bufferizzare uno stream di I/O in quanto ottimizza i tempi di accesso alla risorsa. O comunque li riduce.

In sostanza quello che succede è che, invece di leggere carattere per carattere, si immagazzina in una struttura interna (buffer) una sequenza di caratteri. È come se dovessi studiare 12 ore Programmazione III, e ogni 10 minuti vado a prendere l'acqua dal frigo e riempio un bicchiere. Invece di fare così prendo una bottiglia e me la tengo vicina così ottimizzo i tempi e gli accessi al frigo.

Serializable Interface

Nel caso in cui si dovesse far ricorso a Stream di oggetti complessi, utilizzando ad esempio ObjectOutputStream, è necessario che gli oggetti che viaggiano sul canale/stream, siano conosciuti ad entrambi (chi invia e chi riceve). Questo perché al momento dell'invio sul canale, l'oggetto che viene inviato viene in un certo senso compresso sul canale e quindi perde la sua integrità e la sua forma. Dalla parte del ricevente, se non si sa com'era fatto l'oggetto prima della sua compressione, sarà impossibile ricostruirlo. Ed è qui che entra in gioco la Serializzazione.

L'interfaccia Serializable è una Markable Interface (interfaccia vuota, senza metodi e campi), che permette di scambiare Oggetti complessi (non primitivi) su Stream di dati. Questo è possibile in quanto, quando un oggetto viene serializzato, le informazioni che identificano la sua classe vengono registrate nel flusso serializzato. Tuttavia, la stessa definizione della classe ("file .class") non viene inviata. È responsabilità del sistema che sta deserializzando l'oggetto determinare come individuare e caricare i file di classe necessari.

Un'oggetto serializzato, se contiene al suo interno altri oggetti complessi, anch'essi devono implementare Serializable altrimenti verrà sollevata un'eccezione. In alternativa è possibile marcare questi oggetti come transient, ma facendo così quando l'oggetto verrà inviato, quei campi verranno ignorati e quindi persi.

Programmazione ad Eventi

È un paradigma di programmazione in cui il flusso del programma è determinato da eventi quali: azioni dell'utente (clic del mouse, pressioni di tasti), output di sensori o passaggio di messaggi da altri programmi o thread. La programmazione basata sugli eventi è il paradigma dominante utilizzato nelle interfacce utente, incentrato sull'esecuzione di determinate azioni in risposta all'input dell'utente.

In un'applicazione basata su eventi, generalmente c'è un ciclo principale che ascolta gli eventi e quindi attiva una funzione di callback quando viene rilevato uno di quegli eventi.

Chi ascolta in cerca di eventi da gestire, viene chiamato Listener. Una volta che il listener vede che è stato generato l'evento, lo gestisce chiamando l'event handler opportuno.

Il modello di programmazione ad eventi in sostanza è definito da tre attori:

- La sorgente/source
- L'evento/event
- L'ascoltatore/listener

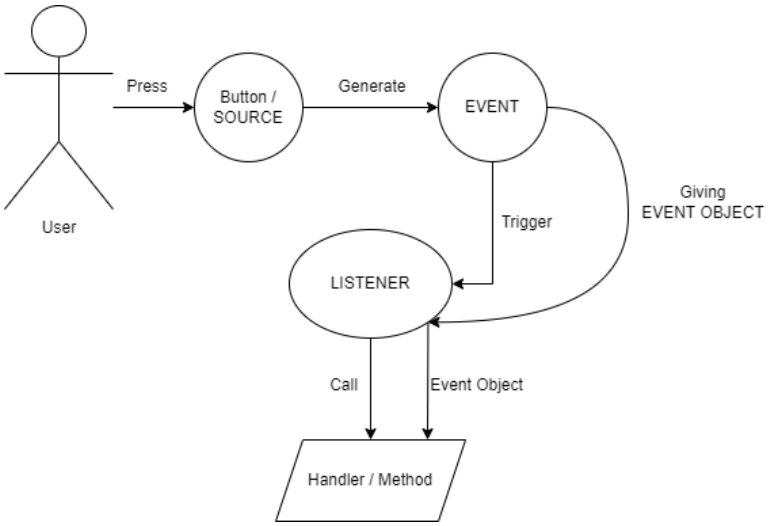
La sorgente (quando esegue un'azione) attiva/genera un evento che viene catturato dal suo ascoltatore/Listener che gestisce l'evento (attraverso un metodo di solito).

In javaFX EventHandler è sinonimo di Listener.

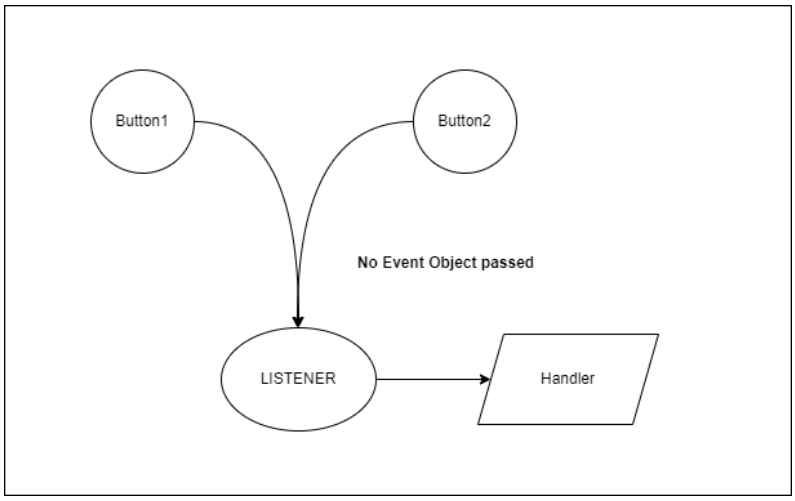
Per ogni tipo di evento è definita una interfaccia che il listener relativo deve implementare (ogni listener può gestire eventi di un certo tipo). Es:

- ActionListener (per eventi da bottoni)
- MouseListener (eventi del mouse)
- MouseMotionListener (spostamenti del mouse)

Gli eventi sono gestiti con un meccanismo di delega: La sorgente, quando genera un evento, passa un oggetto che descrive l'evento ad un "listener" che gestisce l'evento. Il listener deve essere "registrato" presso la sorgente. Il passaggio dell'evento causa l'invocazione di un metodo del listener.



È importante il passaggio dell'Event Object al Listener: immaginiamo uno scenario in cui ci sono due bottoni ascoltati dallo stesso listener ma non viene passato l'Event Object al Listener: In base al bottone premuto vogliamo fare cose diverse, senza passare la sorgente dell'evento avremo:



L'handler non saprebbe quale dei due button ha generato l'evento (L'event object infatti, come detto prima, descrive le caratteristiche dell'evento, tra cui il nome del bottone ad esempio).

In Swing, se volessimo fare una cosa simile:

```
button.addActionListener(new ActionListener() {
		public void actionPerformed(ActionEvent event) {
			if(event.getActionCommand().equals("button1"))
			....
		}
});
```

Questo esempio è fatto con classe anonima, si può fare anche con le lambda. In javaFx:

```
button.addEventHandler(MouseEvent.MOUSE_CLICKED, eventObject -> {
		String text = ((Button)eventObject.getSource()).getText();
		if(text.equals("button1")){
			....
		 }
});
// Oppure
button.setOnAction(eventObject ->{
		String text = ((Button)eventObject.getSource()).getText();
		if(text.equals("button1")){
			....
```

```
        }

});
```

Observer Pattern

Il pattern Observer è un design pattern utilizzato come base architetturale di molti sistemi di gestione di eventi, utilizzato nelle applicazioni software per renderle modulari.

Il modello sostanzialmente si basa su un oggetto (l'oggetto osservabile), denominato soggetto, che mantiene un elenco dei suoi "dipendenti", chiamati osservatori, e notifica loro automaticamente qualsiasi cambiamento di stato, di solito chiamando uno dei loro metodi.

Il pattern Observer viene utilizzato spesso nelle GUI per permettere di disaccoppiare la gestione delle azioni utente dalla gestione delle componenti delle interfacce grafiche. In pratica, si assume che:

- L'applicazione rappresenti il proprio stato in modo esplicito, attraverso opportune variabili
- Le azioni dell'utente sull'interfaccia grafica, catturate dai Listener, cambino lo stato (cioè, il valore delle variabili) dell'applicazione
- L'interfaccia utente osservi lo stato dell'applicazione e, se ci sono cambiamenti, reagisca modificando i dati visualizzati secondo le proprie regole interne

Queste due classi possono essere utilizzate per implementare molto di più della semplice architettura MVC. Sono adatti a qualsiasi sistema in cui è necessario notificare automaticamente agli oggetti le modifiche che si verificano in altri oggetti.

La parte obsoleta di setChanged e notifyObserver è sulle slide, conviene guardarla perché potrebbe uscire nella prova scritta.

Pattern MVC

Il pattern Model-View-Controller è un modello di progettazione del software comunemente utilizzato per lo sviluppo di interfacce utente che dividono la logica del programma in tre elementi interconnessi in modo da aumentare la modularità del software e separare le attività da svolgere:

- **Modello**: Il componente centrale del pattern. È la struttura dati dinamica dell'applicazione, indipendente dall'interfaccia utente. Gestisce direttamente i dati, la logica e le regole dell'applicazione.
- **Vista**: Modella a livello grafico i dati del Modello. Mette a disposizione un'interfaccia grafica all'utente in modo tale da interpretare a livello grafico i dati, mettendo a disposizione metodi per l'interazione tra utente e modello (bottoni, text-area, etc..).
- **Controllore**: Accetta l'input e lo converte in comandi per il modello o la vista.

Oltre a dividere l'applicazione in questi componenti, la progettazione modello-vista-controller definisce le interazioni tra di essi:

- Il modello è responsabile della gestione dei dati dell'applicazione. Riceve le istruzioni dal controller che sua volta riceve l'input dall'utente.
- La vista rende la presentazione del modello in un formato particolare (nel nostro caso un'interfaccia grafica).

- Il controller risponde all'input dell'utente ed esegue interazioni sugli oggetti del modello di dati. Il controller riceve l'input, facoltativamente lo convalida e quindi istruisce il modello in base all'input.

JavaFX

JavaFX è una libreria grafica di java per lo sviluppo di Interfacce grafiche.

I vantaggi nel suo uso sono molteplici:

- API più pulite
- Utilizzo dei fogli di stile per modellare l'aspetto degli oggetti grafici
- Utilizzo di property per legare modello e vista in maniera trasparente
- A differenza di Swing, permette di definire l'interfaccia grafica attraverso XML

Una volta eseguita un'applicazione JavaFX, il metodo principale che viene eseguito è il metodo **start**. Questo metodo riceve come parametro uno Stage (che è la window principale dell'applicazione), a cui poi è possibile appendere una scena (contenitore principale per gli oggetti grafici). Lo stage va associato per forza ad un oggetto di layout come:

- GridPane
- StackPane
- BorderPane
- ...

EventHandlers

Alle componenti grafiche si possono associare dei gestori di eventi. I gestori di eventi (EventHandlers) consentono di gestire gli eventi durante la fase di bubbling degli eventi. Un nodo può avere uno o più gestori per la gestione di un evento. Un singolo gestore può essere utilizzato per più di un nodo. Inoltre, un nodo può registrare più di un tipo di evento.

CSS in JavaFX

JavaFX permette di separare il contenuto di un'interfaccia grafica dal suo aspetto, utilizzando i fogli di stile e quindi assegnando ad oggetti di interfaccia grafica, ID o classi.

Per aggiungere un foglio di stile ad una scena, è possibile utilizzare il metodo di Scene:

```
scene.getStylesheets().add(MyApp.class.getResource("PATH_TO_CSS.css").toExternalForm());
```

Per assegnare un ID ad un elemento dell'interfaccia grafica in modo da applicare lo stile descritto nel file CSS, si utilizza:

- Il metodo setID per assegnare un ID ad un nodo:

  ```
  node.setId("id");
  ```

- Il metodo getStyleClass per assegnare una classe:

  ```
  node.getStyleClass().add("class");
  ```

JavaFX con Java Collection

JavaFX utilizza le classi di Java Collection, estendendole per fornire comportamenti aggiuntivi al fine di implementare le funzionalità di Observer e Observable. Ad esempio, attraverso l'utilizzo di ObservableList e ListView

JavaFXML

Java offre il modo di separare la logica dell'applicazione dalla sua interfaccia grafica attraverso XML. Dal punto di vista del Model View Controller (MVC), il file FXML che contiene la descrizione dell'interfaccia utente è la vista.

Per evitare di scrivere tutto il file XML (e quindi tutta l'interfaccia grafica) a mano è possibile utilizzare un software esterno chiamato SceneBuilder.

Scene Builder è uno strumento di progettazione che genera il codice sorgente FXML dell'interfaccia utente per l'applicazione. SceneBuilder aiuta a creare rapidamente un prototipo per un'applicazione interattiva che collega i componenti alla logica dell'applicazione.

Poiché Scene Builder utilizza XML come formato di serializzazione, il codice FXML prodotto è molto chiaro ed è possibile modificare ulteriormente i file FXML, generati da Scene Builder, in qualsiasi editor di testo o XML

JavaFXML – gestione degli eventi

Il codice sorgente generato da SceneBuilder descrive i componenti grafici ma è assente di logica di gestione dell'interfaccia (quando si interagisce con i componenti, non si modifica nessuno stato).

Il collegamento tra un componente grafico ed il suo comportamento avviene attraverso gli **Event handler attributes**. Gli Event handler attributes sono un mezzo conveniente per allegare comportamenti agli elementi del documento FXML.

Uno dei metodi più opportuni è quello di utilizzare quello che viene chiamato **controller method event handler.** Questo sta ad indicare che un metodo definito nel controller verrà agganciato all'oggetto FXML.

Un gestore di eventi del controller è specificato da un simbolo hash iniziale seguito dal nome del metodo del gestore:

```
<VBox xmlns:fx="http://javafx.com/fxml">
    <children>
        <Button text="Click Me!" onAction="#handleButtonAction"/>
    </children>
</VBox>
```

Così però avremmo definito il metodo che nel controller dovrà gestire l'evento di click del button, ma non abbiamo detto qual è il controller che lo gestirà. Per fare ciò bisogna agganciare il controller alla Root del documento FXML:

```
<VBox fx:controller="com.foo.MyController" xmlns:fx="http://javafx.com/fxml">
    <children>
```

```
        <Button fx:id="button" text="Click Me!" onAction="#handleButtonAction"/>
    </children>
</VBox>
```

Con l'attributo "fx:controller" quindi si associa la classe controller al documento FXML.

Il controller sarà così definito:

```
package com.foo;

public class MyController {

    public Button button;

    public void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
    }
}
```

Da notare che il metodo ed il button definiti hanno visibilità pubblica. Questo di solito non è consigliato. Se si vuole limitare la visibilità al Controller bisogna utilizzare le Java Annotation

Con visibilità privata quindi diventerebbe:

```
package com.foo;

public class MyController {

    @FXML
    private Button button;

    @FXML
    private void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
    }
}
```

L'annotazione javafx.fxml.FXML, rende l'attributo/metodo privato, accessibile al file FXML.

JavaFX – JavaBeans e Properties

JavaFX si basa sul modello JavaBeans ma ampliato e modificato. Le properties di JavaFX sono spesso utilizzate insieme al bindings.

Il bindings è un meccanismo che permette di esprimere relazioni dirette tra variabili: quando oggetti partecipano a delle associazioni/bindings, le modifiche apportate ad uno degli oggetti si riflette sugli altri.

In un'interfaccia utente grafica (GUI) questo concetto è fondamentale poiché mantiene automaticamente la visualizzazione sincronizzata con i dati sottostanti dell'applicazione (appena i dati vengono modificati, la vista si aggiorna di conseguenza in real time).

Come definire una property

```
class Bill {

    // Define a variable to store the property

    private DoubleProperty amountDue = new SimpleDoubleProperty();


    // Define a getter for the property's value
```

```
    public final double getAmountDue(){return amountDue.get();}

    // Define a setter for the property's value

    public final void setAmountDue(double value){amountDue.set(value);}


     // Define a getter for the property itself

    public DoubleProperty amountDueProperty() {return amountDue;}

}
```

Qui abbiamo rappresentato una property chiamata “amountDue” con la convenzione JavaBean.

La differenza tra DoubleProperty e SimpleDoubleProperty è che la prima è la classe astratta che definisce i metodi, la seconda è un’implementazione concreta del tipo di dato.

Da notare che il tipo (DoubleProperty) non è un tipo primitivo ma un Wrapper che incapsula il primitivo ed aggiunge funzionalità extra (tutte le classi che derivano da javafx.beans.property contengono il supporto integrato per l’osservabilità ed il binding):

javafx.beans.property

Class DoubleProperty

java.lang.Object
    javafx.beans.binding.NumberExpressionBase
        javafx.beans.binding.DoubleExpression
            javafx.beans.property.ReadOnlyDoubleProperty
                javafx.beans.property.DoubleProperty

All Implemented Interfaces:

NumberExpression, Observable, Property<Number>, ReadOnlyProperty<Number>, ObservableDoubleValue, ObservableNumberValue, ObservableValue<Number>, WritableDoubleValue, WritableNumberValue, WritableValue<Number>

Threads

Nella programmazione simultanea, ci sono due unità di base di esecuzione: processi e thread.

Nel linguaggio di programmazione Java, la programmazione simultanea riguarda principalmente i thread. I thread sono chiamati processi leggeri. Sia i processi che i thread forniscono un ambiente di esecuzione, ma la creazione di un nuovo thread richiede meno risorse rispetto alla creazione di un nuovo processo.

I thread condividono le risorse del processo, inclusa la memoria e i file aperti: ciò rende la comunicazione efficiente, ma potenzialmente problematica.

Ogni applicazione ha almeno un thread, o più, se si contano i thread di "sistema" che fanno cose come la gestione della memoria e la gestione del segnale. Ma dal punto di vista del

programmatore dell'applicazione, inizi con un solo thread, chiamato thread principale. Questo thread ha la capacità di creare thread aggiuntivi.

Creare e lanciare un thread

Un'applicazione che crea un'istanza di Thread deve fornire il codice che verrà eseguito in quel thread. Ci sono due modi per farlo:

- Definire un oggetto che implementa Runnable
- Definire un oggetto che estende Thread

La differenza sostanziale è che un oggetto che estende Thread non può più estendere alcun'altra classe. Il primo approccio è più flessibile ed applicabile alle API di gestione dei thread di alto livello (Thread Pool).

È possibile assegnare le priorità ai thread (da 1 a 10) con il metodo setPriority: lo scheduler darà priorità ai Thread con priorità più alta.

Per la sua esecuzione, è possibile farla inline

- Se l'oggetto implementa Runnable

```
new Thread(new RunnableClass()).start();
```

- Se l'oggetto estende Thread

```
new MyThread().start();
```

Oppure creando l'istanza e poi avviandola separatamente:

1. Se l'oggetto implementa Runnable:

```
Thread t = new Thread(new RunnableClass());

…

…

t.start();
```

2. Se l'oggetto estende Thread

```
MyThread t = new MyThread();

…

…

t.start();
```

Thread Daemons

Java offre due tipi di thread: thread utente e thread daemon.

I thread utente sono thread ad alta priorità. **La JVM attenderà che qualsiasi thread utente completi la sua attività prima di terminarla.**

D'altra parte, i thread daemon sono thread a bassa priorità il cui unico ruolo è fornire servizi ai thread utente. Poiché i thread daemon sono pensati per servire i thread utente e sono necessari solo mentre i thread utente sono in esecuzione, non impediranno alla JVM di uscire una volta che tutti i thread utente hanno terminato la loro esecuzione.

Ecco perché i loop infiniti, che in genere esistono nei thread daemon, non causeranno problemi, perché qualsiasi codice, inclusi i blocchi finally, non verrà eseguito una volta che tutti i thread utente avranno terminato la loro esecuzione. Per questo motivo, i thread daemon non sono consigliati per le attività di I/O.

Per indicare che un Thread debba o meno comportarsi come un Daemon, si utilizza il metodo setDaemon()

Sincronizzazione tra Thread

I thread comunicano principalmente condividendo l'accesso ai campi e agli oggetti a cui fanno riferimento. Questa forma di comunicazione è estremamente efficiente, ma rende possibili due tipi di errori:

1. **interferenza di thread**: L'interferenza si verifica quando due operazioni, in esecuzione in thread diversi, ma che agiscono sugli stessi dati, si interfogliano. Ciò significa che le due operazioni consistono in più passaggi e le sequenze di passaggi si sovrappongono. Supponiamo di avere questa classe Counter

```
class Counter {

    private int c = 0;


    public void increment() {

        c++;

    }


    public void decrement() {

        c--;

    }


    public int value() {

        return c;

   }

}
```

Se creassimo un'istanza di Counter e la passassimo a due Thread differenti:

- Il primo Thread incrementa il valore di 'c' invocando increment()
- Il secondo Thread decrementa il valore di 'c' invocando decrement()

Poiché le operazioni di incremento e decremento non sono atomiche (queste si traducono in una lista di operazioni JVM), se siamo fortunati il risultato sarà 2, ma se le operazioni dei due thread si interfogliano in modo errato ad esempio:

- Thread A: Preleva c.
- Thread B: Preleva c.
- Thread A: Incrementa il valore prelevato; il risultato è 1.
- Thread B: Decrementa il valore prelevato; Il risultato è -1.
- Thread A: Memorizza il risultato in c; c ora è 1.
- Thread B: Memorizza il risultato in c; c ora è -1.

Da questo ne risulta che il risultato del Thread A è perso, sovrascritto dal Thread B.

2. **errori di consistenza della memoria**: si verificano quando thread diversi hanno visualizzazioni incoerenti di quelli che dovrebbero essere gli stessi dati. Ad esempio, sempre prendendo l'esempio di Counter, se le operazioni fossero:
   - Thread A: increment()
   - Thread B: Stampa il valore di c

   Se le due istruzioni fossero state eseguite nello stesso thread, sarebbe lecito ritenere che il valore stampato sarebbe "1". Ma se le due istruzioni venissero eseguite in thread separati, il valore stampato potrebbe essere "0", perché non vi è alcuna garanzia che la modifica del thread A sia visibile al thread B, a meno che il programmatore non abbia stabilito una relazione accade prima tra questi due affermazioni.

Lo strumento necessario per prevenire questi errori è la sincronizzazione.

Tuttavia, anche utilizzando la sincronizzazione si va in contro a problemi come la contesa tra thread di una qualche risorsa: questo si verifica quando più thread tentano di accedere alla stessa risorsa contemporaneamente e fanno sì che Java esegua uno o più thread più lentamente o addirittura ne sospenda l'esecuzione. Starvation e il livelock sono forme di contesa tra thread.

Syncronized

Java fornisce due metodi di sincronizzazione base:

- Metodi Sincronizzati
- Istruzioni/blocchi sincronizzati

I metodi sincronizzati consentono una strategia semplice per prevenire l'interferenza dei thread e gli errori di coerenza della memoria: se un oggetto è visibile a più di un thread, tutte le letture o le scritture sulle variabili di quell'oggetto vengono eseguite tramite metodi synchronized

Introducendo i metodi sincronizzati alla classe Counter precedentemente definita avremmo:

```
public class SynchronizedCounter {

    private int c = 0;
```

```
    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

- Innanzitutto, non è possibile che due invocazioni di metodi sincronizzati sullo stesso oggetto si intersechino. Quando un thread esegue un metodo sincronizzato per un oggetto, tutti gli altri thread che invocano metodi sincronizzati per lo stesso oggetto bloccano (sospendono l'esecuzione).
- In secondo luogo, quando un metodo sincronizzato esce, stabilisce automaticamente una relazione happens-before con qualsiasi successiva chiamata di un metodo sincronizzato per lo stesso oggetto. Ciò garantisce che le modifiche allo stato dell'oggetto siano visibili a tutti i thread.

Lock Intrinsechi nella Sincronizzazione

La sincronizzazione sopra spiegata si basa più nello specifico su entità interne al linguaggio chiamate Lock (intrinsechi). I blocchi intrinseci svolgono un ruolo in entrambi gli aspetti della sincronizzazione: imporre l'accesso esclusivo allo stato di un oggetto e stabilire relazioni happens-before che sono essenziali per la visibilità.

Ogni oggetto ha un blocco intrinseco ad esso associato. Per convenzione, un thread che necessita di un accesso esclusivo e coerente ai campi di un oggetto deve acquisire il blocco intrinseco dell'oggetto prima di accedervi e quindi rilasciare il blocco intrinseco al termine. Si dice che un thread possieda il blocco intrinseco tra il momento in cui ha acquisito il blocco e il blocco è stato rilasciato (lo rilascia quando il metodo ritorna). Finché un thread possiede un blocco intrinseco, nessun altro thread può acquisire lo stesso blocco. L'altro thread si bloccherà quando tenterà di acquisire il blocco.

Quando un thread rilascia un blocco intrinseco, viene stabilita una relazione happens-before tra quell'azione e qualsiasi successiva acquisizione dello stesso blocco.

La parte dell'happens-before non serve... se volete approfondire.

Cosa succede quando viene invocato un metodo sincronizzato statico? Poiché un metodo statico è associato a una classe, non a un oggetto. In questo caso, il thread acquisisce il blocco intrinseco per l'oggetto Class associato alla classe. Pertanto, **l'accesso**

ai metodi statici della classe è controllato da un blocco distinto dal blocco per qualsiasi istanza della classe.

Blocchi Sincronizzati

Un altro modo per creare codice sincronizzato è con blocchi sincronizzati (la prof le chiama sezioni critiche). A differenza dei metodi sincronizzati, il blocco sincronizzato deve specificare l'oggetto che fornirà (se libero) il lock intrinseco.

```
public void method(Object lock) {

    synchronized(lock) {

        // Istrunctions

    }

}
```

I blocchi sincronizzate sono utili anche per migliorare la granularità della concorrenza.

Supponiamo di avere questo scenario:

```
public class foo {

    private long c1 = 0;

    private long c2 = 0;


    public syncronized void inc1() {

            c1++;

    }

    public syncronized void inc2() {

            c2++;

    }}
```

Se un Thread esegue mille incrementi sulla variabile c1, e un altro Thread esegue un incremento di mille sulla variabile c2, finchè il primo Thread non ha completato i mille incrementi, il secondo Thread non avrà il Lock e non potrà lavorare su c2. Se c1 e c2 sono logicamente separate, non ha molto senso far aspettare il secondo Thread.

In questo caso è utile creare due oggetti che fungono da Lock per le due variabili e quindi usare i blocchi sincronizzati

```
public class MsLunch {

    private long c1 = 0;

    private long c2 = 0;
```

```
    private Object lock1 = new Object();

    private Object lock2 = new Object();


    public void inc1() {

        synchronized(lock1) {

            c1++;

        }

    }


    public void inc2() {

        synchronized(lock2) {

            c2++;

        }

    }

}
```

Blocchi custoditi (Guarded Block)

Il blocco custodito/blocco protetto è un altro meccanismo per coordinare l'esecuzione di più thread in un ambiente multithread. Il blocco continua a verificare che una particolare condizione diventi vera e solo in quel caso riprende l'effettiva esecuzione del thread.

Supponiamo, ad esempio un metodo, take, che rimuove un elemento da una Queue<T>. Un tale metodo potrebbe, in teoria, semplicemente eseguire un ciclo finché la condizione (che verifica che ci siano elementi) non è soddisfatta, ma quel ciclo è dispendioso, poiché viene eseguito continuamente durante l'attesa.

```
public T take() throws InterruptedException {

  while(queue.isEmpty()) {

  }

  T item = queue.remove();

  return item;

}
```

Un metodo più efficiente utilizza Object.wait() che sospende il Thread corrente.
L'invocazione del metodo wait non ritorna finchè un altro Thread non ha emesso una notifca invocando il metodo Notify o NotifyAll.

```
public synchronized void put(T element) throws InterruptedException {
    while(queue.size() == MAX_CAPACITY) {
        wait();
    }
    queue.add(element);
    notify(); // notifyAll() for multiple producer/consumer threads
}

public synchronized T take() throws InterruptedException {
    while(queue.isEmpty()) {
        wait();
    }
    T item = queue.remove();
    notify(); // notifyAll() for multiple producer/consumer threads
    return item;
}
```

Da notare che i metodi sono Syncronized. Questo è dovuto al fatto che, in caso di assenza della keyword syncronized si potrebbero verificare quelli che vengono chiamati "segnali mancati" o "notifiche mancate"

Supponiamo che la keyword syncronized non ci sia, e quindi non si utilizzino i lock intrinsechi.

- un primo Thread che produce (produttore) chiama put() quando la coda è piena, quindi controlla la condizione, vede che la coda è piena. Tuttavia, prima che possa bloccare un altro thread è stato avviato (un consumatore).
- Questo secondo thread (consumatore) chiama take(), rimuove un elemento della coda e notifica (quindi la coda non è più piena). Tuttavia, poiché il primo thread ha già verificato la condizione, chiamerà semplicemente wait, anche se potrebbe fare progressi.

Eseguendo la sincronizzazione su un oggetto condiviso, è possibile assicurarsi che questo problema non si verifichi, poiché la take() chiamata del secondo thread non sarà in grado di procedere finché il primo thread non sarà stato effettivamente bloccato.

Java Pipes

Le pipes di Java.io offre la possibilità di comunicare tra due thread in esecuzione nella stessa JVM. Pertanto, le pipe possono anche essere sorgenti o destinazioni di dati.

Non è possibile utilizzare una pipe per comunicare con un thread in una JVM diversa (processo diverso). Il concetto di pipe in Java è diverso dal concetto di pipe in Unix/Linux, in cui due processi in esecuzione in spazi di indirizzi diversi possono comunicare tramite una pipe. In Java, le parti in comunicazione devono essere in esecuzione nello stesso processo e devono essere thread diversi.

Con le pipe, non è necessario sincronizzare esplicitamente produttore e consumatore perché la sincronizzazione avviene in automatico man mano che scrivono e leggono sul/dal canale di comunicazione.

Esempio di pipe. Produttore:

```
class Producer extends Thread {
    private Random rand = new Random();
    private PipedWriter out = new PipedWriter();

    public PipedWriter getPipedWriter() {
        return out;
    }

    public void run() {
        while (true) {
            try {
                int num = rand.nextInt();
                out.write(num);
                out.flush();
                sleep(Math.abs(rand.nextInt() % 1000));
            } catch (Exception e) {
                System.out.println("Error: " + e);
            }
        }
    }
}
```

Consuamatore:

```
class Receiver extends Thread {
    private PipedReader in;

    public Receiver(Producer producer) throws IOException {
        in = new PipedReader(producer.getPipedWriter());
    }

    public void run() {
        try {
            while (true) {
                System.out.println("Read: " + in.read());
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Un ipotetico main sarebbe:

```
class PipedIO {
    public static void main(String[] args) throws Exception {
        Producer sender = new Producer();
        Receiver receiver = new Receiver(sender);
        sender.start();
        receiver.start();
    }
}
```

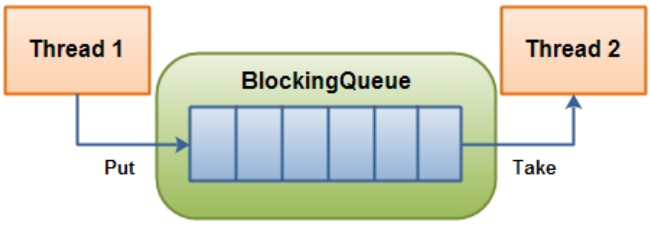
Da notare che prima si crea il produttore che crea la PipeStream, successivamente, il consumatore crea il suo canale agganciandosi allo stream del produttore.

Utilizzando PipedWriter e PipedReader, è possibile solo leggere singoli caratteri alla volta. Per poter lavorare con oggetti complessi è necessario utilizzare ObjectOutputStream e PipedOutputStream tipo:

```
PipedOutputStream pipedOutputStream = new PipedOutputStream();

PipedInputStream pipedInputStream = new PipedInputStream();

//

ObjectOutputStream objectOutputStream = new ObjectOutputStrem(pipedOutputStream)

ObjectInputStream objectInputStream = new ObjectInputStream(pipedInputStream);
```

Blocking Queue Interface

L'interfaccia Java BlockingQueue rappresenta una coda thread-safe in cui inserire e prelevare elementi. In altre parole, più thread possono inserire e prelevare elementi contemporaneamente da un Java BlockingQueue, senza che si verifichino problemi di concorrenza. Le implementazioni di BlockingQueue sono progettate per essere utilizzate principalmente per le code produttore-consumatore.



Il thread di produttore continuerà a produrre nuovi oggetti e li inserirà nella BlockingQueue, fino a quando la coda non raggiunge un limite superiore su ciò che può contenere. Se la coda di blocco raggiunge il limite superiore, il thread produttore viene bloccato (messo in wait) durante il tentativo di inserire il nuovo oggetto. Rimane bloccato fino a quando un thread che consuma non rimuove un oggetto dalla coda.

Il thread consumatore continua a prelevare oggetti dalla BlockingQueue per elaborarli. Se il thread consumatore tenta di estrarre un oggetto da una coda vuota, questo viene bloccato (messo in wait) finché un thread di produzione non inserisce un oggetto nella coda.

```
BlockingQueue bq = new ArrayBlockingQueue(5); // Coda con capacità 5
```

Lock interface e condition

Java oltre ad offrire dei lock impliciti attraverso l'uso di blocchi/metodi sincronizzati, offre un'interfaccia che permette di implementare Lock espliciti. L'interfaccia Lock tra i metodi principali offre quelli per l'acquisizione ed il rilascio di un lock:

- lock(): Acquisisce il lock. Se il blocco non è disponibile, il thread corrente viene disabilitato per scopi di pianificazione dei thread e rimane inattivo fino a quando il blocco non viene acquisito.
- unlock(): Rilascia il lock

ReentrantLock è un'implementazione dell'interfaccia Lock.

Il ReentrantLock è appunto un oggetto che permette la sincronizzazione di blocchi di istruzioni proprio come il Lock implicito attraverso l'uso di Synchronized ma con funzionalità estese. Le capacità estese includono:

- La possibilità di avere più di una variabile condizionale per monitor. I monitor che utilizzano la parola chiave sincronizzata possono averne solo una. Ciò significa che i blocchi rientranti supportano più di una coda wait()/notify().
- La capacità di rendere il lock "fair". I blocchi sincronizzati sono "unfair".
- La possibilità di verificare se il blocco è stato bloccato.
- La possibilità di ottenere l'elenco dei thread in attesa del blocco.

Il ReentrantLock appunto può avere più di una variabile condizionale, grazie all'uso di oggetti Condition. Una volta creato il lock, a quel lock è possibile assegnare più variabili condizionali grazie al metodo newCondition() che restituisce una Condition.

Una variabile di condizione è un costrutto che fornisce operazioni di attesa e notifica e che mantiene una serie di thread in attesa.

Un predicato di condizione è un predicato (una funzione o un'espressione con valore booleano) chiamato dal codice che utilizza una variabile di condizione. In breve, un thread attende su una variabile di condizione finché il predicato non è true e un thread notifica (o segnala) la variabile di condizione quando il predicato diventa true.

Esempio:

```
final Lock lock = new ReentrantLock();

final Condition producers = lock.newCondition(); // VARIABILE DI CONDIZION

final Condition consumers = lock.newCondition(); // VARIABILE DI CONDIZION



public void put(T element) throws InterruptedException {

  lock.lock();

  try {
```

```
        while(queue.size() == MAX_CAPACITY) { // PREDICATO DI CONDIZIONE
            producers.await();
        }
        queue.add(element);
        consumers.signal();
    } finally {
        lock.unlock();
    }
}

public T take() throws InterruptedException {
    lock.lock();
    try{
        while(queue.isEmpty()) { // PREDICATO DI CONDIZIONE
            consumers.await();
        }
        T item = queue.remove();
        producers.signal();
        return item;
    } finally {
        lock.unlock();
    }}
```

A differenza della classica wait e notify (con una singola variabile di condizione), qui abbiamo due code di attesa e di sblocco. Così quando un consumatore consuma una risorsa, risveglia solo i produttori (e viceversa). **Nella wait notify con blocchi sincronizzati (dove c'è una singola variabile di condizione) nel caso in cui un consumatore consumava una risorsa, risvegliava tutti, sia consumatori che produttori (con la notifyAll).**

Per quanto riguarda la politica del "fair", i blocchi sincronizzati non si preoccupano se c'è un Thread che aspetta da tempo di passare. Con i Lock espliciti è possibile definire una politica "fair":

```
Lock fairLock = new ReentrantLock(true);
```

Come da documentazione

ReentrantLock

```
public ReentrantLock(boolean fair)
```

Creates an instance of `ReentrantLock` with the given fairness policy.

Parameters:

`fair` - true if this lock should use a fair ordering policy

ReadWriteLock

Come Lock, questa è solo un'interfaccia, il tipo concreto che la implementa si chiama ReentrantReadWriteLock (dogshit java naming convention).

Un java.util.concurrent.locks.ReadWriteLock è uno strumento di blocco dei thread di alto livello. Consente a vari thread di leggere una risorsa specifica ma consente solo a uno di scriverla alla volta.

L'approccio è che più thread possono leggere da una risorsa condivisa senza causare errori di concorrenza.

Regole:

Nella struttura è presente un lock in lettura ed un lock in scrittura che consentono a un thread di bloccare ReadWriteLock in lettura o in scrittura.

- **Blocco di lettura:** se nessun thread ha richiesto il lock di scrittura, più thread possono bloccare il lock di lettura. Significa che più thread possono leggere i dati in questo preciso momento, purché non ci siano thread per scrivere i dati o per aggiornare i dati.
- **Blocco scrittura:** se nessun thread sta scrivendo **o leggendo**, solo un thread alla volta può bloccare il lock per la scrittura. Gli altri thread devono attendere fino al rilascio del lock. Significa che solo un thread può scrivere i dati al momento e gli altri thread devono attendere.

Da notare che questo approccio **da solo** non è consistente: In un contesto di produttori-consumatori, se avessimo tanti produttori e pochi consumatori, avremmo tanti put e pochi take, fino ad avere la coda piena. A questo punto se altri produttori provano ad inserire elementi in coda, ma la coda è piena, o fanno un ciclo infinito finché non la coda non ha un posto, oppure implementa il meccanismo di Conditions (che è la scelta migliore).

Thread Pools

Sappiamo che ci sono due modi per creare un thread in Java (estendendo Thread o implementando Runnable). Gli oggetti thread utilizzano una quantità significativa di memoria e, in un'applicazione su larga scala, l'allocazione e la deallocazione di molti oggetti thread crea un notevole sovraccarico di gestione della memoria.

java.util.concurrent.Executors fornisce metodi per creare ThreadPool di thread di lavoro. I pool di thread risolvono questo problema mantenendo attivi i thread e riutilizzandoli. Tutte le attività in eccesso che possono essere gestite dai thread nel pool vengono mantenute in una

coda. Una volta che uno qualsiasi dei thread si libera, riprende l'attività successiva da questa coda.

L'uso dei ThreadPool in sostanza riduce al minimo l'overhead dovuto alla creazione del thread.

Quello che fa il ThreadPool è quello di creare un insieme di Thread che rimangono attivi. Dopo che un thread del pool di thread ha terminato l'esecuzione dell'attività, ritorna al pool di thread (non viene distrutto), in modo che possa prelevare la successiva attività disponibile.

Inoltre, grazie ai ThreadPools è possibile limitare il numero di Thread attivi che gestiscono le attività all'interno del pool. Senza un pool di thread o qualche altro meccanismo di limitazione, il numero di thread attivi potrebbe aumentare senza limiti, causando infine l'interruzione del processo da parte del sistema operativo o il blocco del sistema operativo.

Il package java.util.concurrent definisce tre interfacce di Executors:

- Executor: una semplice interfaccia che supporta l'avvio di nuove attività.
- ExecutorService: una sottointerfaccia di Executor, che aggiunge funzionalità che aiutano a gestire il ciclo di vita, sia delle singole attività che dell'esecutore stesso.
- ScheduledExecutorService: una sottointerfaccia di ExecutorService, supporta l'esecuzione futura e/o periodica di attività.

Il modo più semplice per creare un Executor che utilizza pool di Thread consiste nell'invocare i metodi factory di Executors:

- newFixedThreadPool: crea un executor con un pool di Thread fisso.
- newCachedThreadPool: crea un executor con un pool di thread espandibile. Questo esecutore è adatto per applicazioni che avviano molte attività di breve durata.
- newSingleThreadExecutor: crea un executor che esegue una singola attività alla volta.
- ScheduledExecutorService: crea un pool thread con scheduling delle attività (serve per creare pool di thread che eseguiranno i task con un ritardo specificato in input, oppure ciclicamente)

Esempio di fixed:

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

Per assegnare un Runnable all'executor, l'interfaccia Executor offre il metodo execute che esegue il runnable passato, ad un certo punto, nel futuro, secondo la policy specificata.

```
Runnable myRunnable = new MyRunnable();

fixedPool.execute(myRunnable);

//oppure

fixedPool.sumbit(MyRunnable);
```

Differenze:

- submit() può accettare sia Runnable che Callable ma execute() può accettare solo Runnable.

- Il metodo submit() è dichiarato nell'interfaccia ExecutorService mentre il metodo execute() è dichiarato nell'interfaccia Executor .
- Il tipo restituito del metodo submit() è un oggetto Future ma il tipo restituito del metodo execute() è void.

Callable interface e Future Interface

Come detto anche prima, esistono due modi per creare thread: uno estendendo la classe Thread e l'altro creando un thread con Runnable. Tuttavia, una caratteristica che manca in Runnable è che non possiamo fare in modo che un thread restituisca un risultato quando termina, cioè quando run() viene completato. Per supportare questa funzione, l'interfaccia Callable ci viene in aiuto.

A differenza di Runnable, Callable è parametrizzato, infatti è così definito

java.util.concurrent

Interface Callable<V>

Type Parameters:

V - the result type of method `call`

Ed offre il metodo Call() che, a differenza appunto di Runnable, ritorna un valore che è lo stesso tipo del parametro di classe

Method Summary

Methods

Modifier and Type	Method and Description
V	`call()` Computes a result, or throws an exception if unable to do so.

Quindi implementando Callable è possibile definire un task che ritorna un valore:

```
public class RandomTask implements Callable<Integer> {

        public Integer call() throws InvalidParamaterException {

                return Math.random();

        }

}
```

Utilizzando solo Callable, non è possibile reperire il valore che il task ritorna. C'è bisogno di una struttura di appoggio.

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

RandomTask t = new RandomTask();
```

```
executor.submit(t);

// Una volta eseguito, come recuperiamo il valore che ritorna? Executor non ha
strutture dati interne che immagazzinano i valori di ritorno.
```

L'interface Future offre metodi per la verifica della terminazione di un Callable e per l'estrazione del suo risultato:

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

RandomTask t = new RandomTask();

Future<Integer> future = executor.submit(t);

// Oppure per splittare dichiarazioni

FutureTask ft = new FutureTask<>(t);

executor.submit(t);
```

In questo modo l'oggetto Future immagazzinerà il risultato appena questo sarà disponibile.

Per sapere, ad un punto del codice, se il risultato è disponibile, si usa il metodo isDone();

il metodo isDone() Restituisce true se l'attività è stata completata. Il completamento può essere dovuto alla normale risoluzione, a un'eccezione o all'annullamento: in tutti questi casi, questo metodo restituirà true.

Se l'attività è terminata, è possibile reperire il valore con il metodo get() che Attende, se necessario, il completamento del calcolo, quindi ne recupera il risultato. Ciò significa che è bloccante: se al momento della chiamata, il task non è completato, il chiamante aspetterà.

```
…

…

// Somewhere else in the code

try {

        int result = (t.get()).intValue();

} catch (Exception e) {

        System.out.println(e.getMessage());

}
```

Java Socket

Java fornisce una raccolta di classi e interfacce che si occupano dei dettagli di comunicazione di basso livello tra il client e il server. Questi sono per lo più contenuti nel pacchetto *java.net*.

Un meccanismo simile è quello delle Pipe, che permette a due Thread in locale di comunicare e scambiarsi dati. Quello che fa il Socket è quello di creare sempre un canale di comunicazione, ma uscire dal contesto local interfacciando due utenti sulla rete.

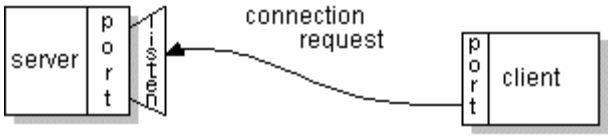
Un socket quindi è un endpoint di comunicazione (un endpoint è una combinazione di un indirizzo IP e un numero di porta. Ogni connessione TCP può essere identificata in modo univoco dai suoi due endpoint) bidirezionale tra due programmi in esecuzione sulla rete. Un socket è associato a un numero di porta in modo che il livello TCP possa identificare l'applicazione a cui i dati sono destinati a essere inviati.

Funzionamento lato server

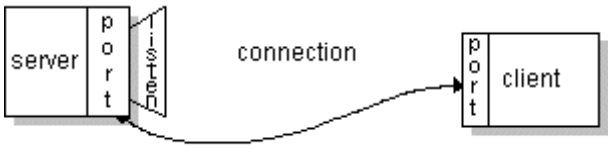
Normalmente, un server viene eseguito su un computer specifico ed aprendo un socket associato a un numero di porta. Il server attende semplicemente, ascoltando sul socket aperto, affinché un client effettui una richiesta di connessione.

Funzionamento lato client

Il client conosce il nome della macchina (IP o nome logico) su cui è in esecuzione il server e il numero di porta su cui quel servizio è in ascolto. Per effettuare una richiesta di connessione, il client tenta di connettersi all'indiritto e sulla porta del server.



Se tutto va bene, il server accetta la connessione. Dopo l'accettazione, il server ottiene un nuovo socket associato all'endpoint remoto impostato sull'indirizzo e sulla porta del client.



Sul lato client, se la connessione viene accettata, viene creato correttamente un socket e il client può utilizzare il socket per comunicare con il server.

A questo punto il client e il server possono comunicare scrivendo o leggendo dai loro socket.

Aiuti e immagini:

https://docs.oracle.com/javase/tutorial/