

JAVA DA ZERO AD UN PROGETTO



PIETRO PASSANTINI

JAVA DA ZERO AD UN PROGETTO

PIETRO PASSANTINI

*****ebook converter DEMO Watermarks*****

SOMMARIO

[Prefazione](#)

[Introduzione a Java](#)

[Introduzione](#)

[La storia](#)

[Le caratteristiche](#)

[La portabilità](#)

[La Java Virtual Machine](#)

[La Memoria Virtuale](#)

[Installazione di Java](#)

[Concetti fondamentali della programmazione](#)

[Struttura di un programma](#)

[Tipi di dato in Java](#)

[I tipi primitivi](#)

[Tipi di dato per riferimento](#)

[I tipi di dato impliciti](#)

[Il primo programma Hello World](#)

[Gli operatori](#)

[Gli operatori bitwise](#)

[Gli operatori ternari](#)

[Le condizioni](#)

[I Cicli](#)

[Controllo del flusso di un ciclo](#)

[Gestione dell'Input e Output da Console](#)

Il Metodo Main

[Esercizio 1](#)

[Esercizio 2](#)

[Esercizio 3](#)

[Esercizio 4](#)

Funzioni

[Le Funzioni](#)

[I parametri](#)

[Le funzioni ricorsive](#)

[Esercizio 5](#)

[Esercizio 6](#)

Array e collezioni

[Come dichiarare e utilizzare array](#)

[Array multidimensionali](#)

[ArrayList](#)

[LinkedList](#)

[HashMap](#)

[HashSet](#)

[Esercizio 7](#)

[Esercizio 8](#)

[Esercizio 9](#)

[Gestione delle eccezioni](#)

Come gestire le eccezioni in Java

[Lancia e cattura](#)

[Finally](#)

[Stack trace](#)

Programmazione orientata agli oggetti

[Concetti di classe e oggetto](#)

[Incapsulamento](#)

[Ereditarietà](#)

[Polimorfismo](#)

[Interfaccia](#)

[Esercizio 10](#)

[Esercizio 11](#)

[Esercizio 12](#)

[Introduzione ai Design Pattern](#)

[Definizione di Design Pattern](#)

[Abstract Factory](#)

[Builder](#)

[Factory Method](#)

[Prototype](#)

[Singleton](#)

[Adapter](#)

[Composite](#)

[Decorator](#)

[Facade](#)

[Proxy](#)

[Observer](#)

[Scelta del pattern appropriato](#)

[Gestione dei File in Java](#)

[Introduzione](#)

[Apertura di un File di Testo](#)

[Random Access File](#)

[File di properties](#)

[Lettura di un File di Proprietà](#)

[Scrittura in un File di Proprietà](#)

[JDBC e SQL](#)

[Introduzione alla Java Database Connectivity](#)

[Flusso Tipico di Uso di JDBC](#)

[Statement e PreparedStatement](#)

[PreparedStatement](#)

[Oggetto ResultSet](#)

[Le transazioni](#)

[Best practice con JDBC](#)

[Problemi di sicurezza SQL injection](#)

[Come funziona SQL Injection](#)

[Esercizio 13: Lettura di Dati da un Database](#)

[Esercizio 14: Inserimento di Dati in un Database](#)

[Esercizio 15: Aggiornamento e Cancellazione di Dati](#)

[I Thread](#)

[La concorrenza](#)

[Esercizio 1: Contatore Condiviso](#)

[Esercizio 2: Produttore-Consumatore](#)

[Esercizio 3: Lettori e Scrittori](#)

[Comunicazione tramite Socket in Java](#)

[Concetti Fondamentali delle Socket](#)

[Creazione di una Socket Client](#)

[Creazione di una Socket Server](#)

[Esempio di Applicazione Client-Server](#)

[Sistema di chat client Server che conserva i messaggi](#)

[Requisiti utente](#)

[Analisi funzionale](#)

[Il protocollo di comunicazione](#)

[Le enumeration](#)

[Diagramma delle classi](#)

[Sviluppo del codice](#)

[Le annotation](#)

[Riferimenti](#)

PREFAZIONE

Quando parliamo di Java parliamo di un mondo veramente ampio, infatti per la sua versatilità Java è utilizzato in una vasta gamma di settori, tra cui lo sviluppo di applicazioni per il web, la creazione di app mobili, lo sviluppo di software per il back-end e la creazione di applicazioni per l'Internet delle cose (IoT). Inoltre, Java è ampiamente utilizzato in campo accademico per insegnare i concetti di programmazione e informatica.

Lo scopo di questo libro è fornire la chiave per poter comprendere e specializzarsi in questo mondo.

L'orientamento di questo corso e quello di apprendere piccole nozioni per volta e metterle subito in pratica, con esempi ed esercizi proposti, fino alla progettazione e lo sviluppo di un progetto reale.

L'autore **Pietro Passantini** è un programmatore e consulente ormai da molti anni nel settore dell'information Technologies, ha seguito lo sviluppo e l'evoluzione di Java sin dalle fasi primordiali, è certificato Java sulla versione 1.4, inoltre è impegnato da molti anni nella formazione in azienda di nuove risorse,

Ringraziamenti ringrazio mia moglie Lidia da sempre il mio punto di appoggio, Fernando Rondoni e Claudio Castagna di Valueson, che mi supportano da anni nel difficile ma gratificante compito di formare nuove risorse.

INTRODUZIONE A JAVA

Introduzione

Java è un linguaggio di programmazione sviluppato negli anni '90 da Sun Microsystems, che successivamente è stata acquisita da Oracle.

Java è noto per la sua portabilità, ovvero la capacità di essere eseguito su diverse piattaforme hardware e software, grazie alla sua caratteristica di compilazione in byte code.

Per sviluppare programmi Java, è necessario installare il JDK (Java Development Kit), che comprende l'ambiente di sviluppo integrato (IDE) e le librerie di sistema.

LA STORIA

Java è stato sviluppato da **James Gosling** e dal suo team di sviluppatori alla Sun Microsystems (acquisita poi da Oracle) alla fine degli anni '80.

L'obiettivo iniziale era quello di creare un linguaggio di programmazione per controllare e gestire i dispositivi elettronici all'interno di casa, come ad esempio i televisori.

Nel 1991 il progetto venne ufficialmente denominato Java e nel 1995 venne rilasciata la prima versione pubblica di Java, denominata Java 1.0. Questa prima versione si concentrava principalmente sulla creazione di applet, ovvero programmi che potevano essere eseguiti all'interno di un browser web. In questo modo, Java ha permesso di creare contenuti interattivi sul web, rivoluzionando il mondo della programmazione.

Nel corso degli anni, Java è diventato sempre più popolare e ha trovato sempre più applicazioni, grazie alle sue caratteristiche di portabilità e sicurezza. Java è stato utilizzato per sviluppare applicazioni desktop, applicazioni web, applicazioni mobile, videogiochi e molto altro ancora.

Nel 2006 Sun Microsystems ha reso Java open source, rendendo il codice sorgente di Java disponibile a tutti gli sviluppatori. Nel 2010 Sun Microsystems è stata acquisita da Oracle, che attualmente gestisce lo sviluppo e la distribuzione di Java.

Oggi Java è uno dei linguaggi di programmazione più popolari e utilizzati al mondo, con una vasta comunità di sviluppatori e una grande varietà di librerie e framework disponibili per semplificare la programmazione Java.

LE CARATTERISTICHE

Le caratteristiche di Java sono:

1. Orientato agli oggetti: Java è un linguaggio di programmazione completamente orientato agli oggetti, il che significa che tutti i dati e le funzioni sono modellati come oggetti.
2. Portabilità: il bytecode generato dal compilatore Java è indipendente dalla piattaforma, il che significa che i programmi Java possono essere eseguiti su qualsiasi sistema che ha una JVM (Java Virtual Machine) installata.
3. Sicurezza: Java è stato progettato per essere un linguaggio di programmazione sicuro. Le applicazioni Java sono eseguite all'interno di un ambiente controllato (la JVM) che limita l'accesso alle risorse del sistema.
4. Garbage collection: Java dispone di un sistema di garbage collection integrato che automaticamente libera la memoria non utilizzata dal programma.
5. Multithreading: Java supporta la creazione di programmi multithreading, ovvero programmi che possono eseguire più attività contemporaneamente.
6. Ampia libreria di classi: Java offre una vasta gamma di librerie di sistema predefinite che forniscono funzionalità per la gestione dei file, il networking, la grafica e molti altri aspetti della programmazione.
7. Sintassi semplice e leggibile: la sintassi di Java è facile da leggere e scrivere, il che lo rende un linguaggio di programmazione molto accessibile anche per i principianti.
8. Comunità attiva: Java ha una grande comunità di sviluppatori attivi che continuano a sviluppare librerie e framework per semplificare la programmazione Java.

Queste sono solo alcune delle caratteristiche del linguaggio di programmazione Java, che lo rendono uno dei linguaggi più popolari e utilizzati al mondo.

LA PORTABILITÀ

Java è noto per la sua portabilità, ovvero la capacità di essere eseguito su diverse piattaforme hardware e software, grazie alla sua caratteristica di compilazione in bytecode.

Il **bytecode** di Java è il formato di codice intermedio ottenuto dalla compilazione del codice sorgente Java. Questo bytecode è indipendente dalla piattaforma, ovvero può essere eseguito su qualsiasi sistema operativo che abbia una macchina virtuale Java (JVM) installata.

La JVM è un software che esegue il bytecode di Java traducendolo in istruzioni specifiche per la piattaforma su cui è in esecuzione, in modo che il programma possa essere eseguito correttamente. Quando si avvia un programma Java, la JVM carica il bytecode e lo esegue, interagendo con il sistema operativo per eseguire le operazioni richieste dal programma.

La compilazione in bytecode offre numerosi vantaggi, come la portabilità, la sicurezza e l'efficienza. Inoltre, il bytecode può essere analizzato e ottimizzato ulteriormente da un compilatore just-in-time (JIT), che converte il bytecode in codice nativo della piattaforma durante l'esecuzione del programma, migliorando le prestazioni complessive.

LA JAVA VIRTUAL MACHINE

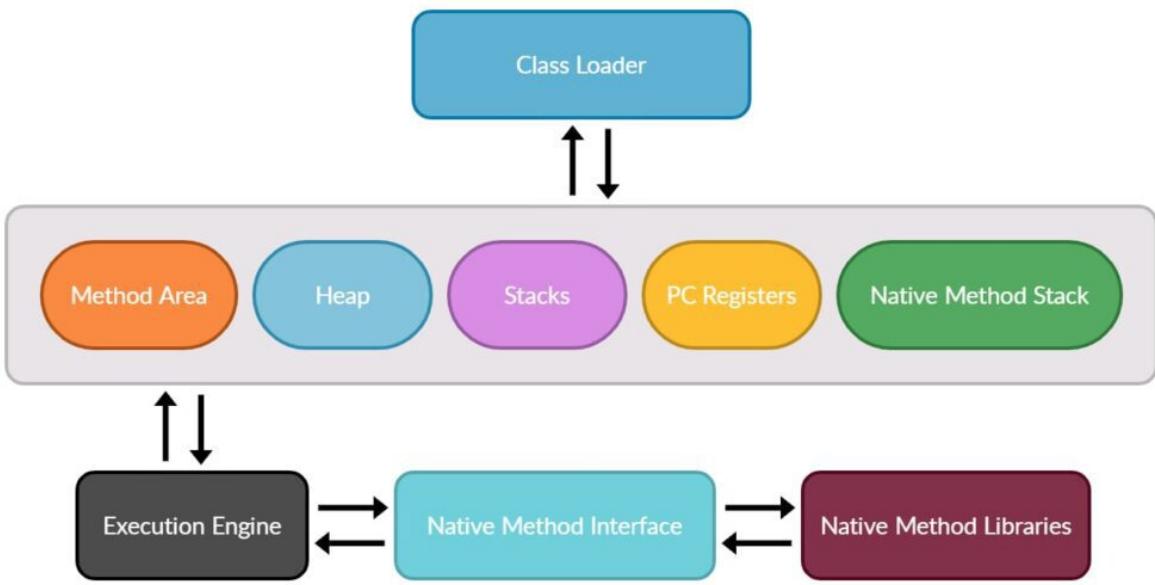
La **JVM** (Java Virtual Machine) è un software che consente l'esecuzione di programmi scritti in linguaggio di programmazione Java. Quando si esegue un programma Java, il codice sorgente viene compilato in bytecode, un formato di codice intermedio che è indipendente dalla piattaforma. La JVM è quindi responsabile di eseguire il bytecode, convertendolo in istruzioni specifiche per la piattaforma sottostante.

Il processo di esecuzione del bytecode da parte della JVM è composto da diverse fasi. Inizialmente, la JVM carica il bytecode in memoria e lo verifica per assicurarsi che sia corretto e sicuro da eseguire.

Successivamente, la JVM interpreta il bytecode, ovvero lo traduce in istruzioni specifiche per la piattaforma sottostante, utilizzando un'interprete integrato.

Durante l'esecuzione del programma, la JVM esegue anche il garbage collection, che consiste nel liberare la memoria non utilizzata dal programma, migliorando così l'efficienza e prevenendo i problemi di memory leak. Inoltre, la JVM supporta anche la compilazione just-in-time (JIT), ovvero la conversione del bytecode in codice nativo della piattaforma, che può migliorare le prestazioni complessive del programma.

Infine, la JVM fornisce un'ampia gamma di librerie di sistema, che offrono funzionalità per la gestione dei file, il networking, l'input/output e molti altri aspetti della programmazione. Grazie alla JVM, i programmi Java possono essere eseguiti su qualsiasi piattaforma che ha una JVM installata, senza la necessità di modificare il codice sorgente originale.



il Class Loader è un meccanismo che carica le classi Java all'interno della Java Virtual Machine (JVM) durante l'esecuzione di un programma. Il compito principale del Class Loader è quello di individuare le classi richieste dal programma, caricarle nella memoria della JVM e prepararle per l'esecuzione.

Ci sono tre tipi di Class Loader in Java: il Bootstrap Class Loader, l'Extension Class Loader e l'Application Class Loader. Il Bootstrap Class Loader è il primo ad essere caricato all'avvio della JVM e si occupa di caricare le classi fondamentali di Java, come ad esempio `java.lang.Object`. L'Extension Class Loader carica le classi di estensione e il loro codice sorgente, mentre l'Application Class Loader carica le classi specifiche dell'applicazione.

In Java, il Class Loader implementa anche il concetto di "sandboxing" (o "scatola di sabbia"), ovvero un ambiente di esecuzione controllato e isolato, che protegge il sistema operativo da eventuali danni causati da programmi malevoli o inaffidabili. In pratica, il Class Loader controlla le risorse alle quali una classe Java può accedere, proteggendo così il sistema da eventuali attacchi o intrusioni.

Inoltre, il Class Loader di Java supporta anche il concetto di "dynamic class loading" (caricamento dinamico delle classi), che permette di caricare le

classi solo quando sono effettivamente necessarie, migliorando le prestazioni e riducendo il consumo di memoria della JVM.

La JVM gestisce la memoria in modo autonomo e automatico, utilizzando un meccanismo denominato "garbage collection". In Java, la memoria è divisa in tre aree principali: heap, stack e area per le costanti.

L'heap è l'area di memoria dove vengono allocate le istanze degli oggetti. L'allocazione dell'heap è gestita dalla JVM e viene effettuata dinamicamente a seconda delle necessità dell'applicazione. L'heap è inoltre soggetto al meccanismo di garbage collection, che rimuove automaticamente gli oggetti non più utilizzati dalla memoria.

Lo stack, invece, è l'area di memoria dove vengono allocate le variabili locali e le informazioni sullo stato delle chiamate ai metodi. Lo stack viene gestito dalla JVM e viene deallocated automaticamente alla fine dell'esecuzione del metodo.

L'area per le costanti è l'area di memoria dove vengono allocate le costanti utilizzate nel programma. Questa area viene gestita staticamente dalla JVM e non può essere modificata durante l'esecuzione del programma.

LA MEMORIA VIRTUALE

La gestione della memoria da parte della JVM avviene tramite il meccanismo di garbage collection. Il garbage collector identifica gli oggetti non più referenziati all'interno del programma e li rimuove automaticamente dalla memoria. In questo modo, la JVM garantisce un utilizzo efficiente della memoria e previene eventuali problemi di memory leak o di esaurimento della memoria.

Inoltre, la JVM fornisce anche la possibilità di configurare le impostazioni di memoria attraverso i parametri di avvio del programma, ad esempio impostando la dimensione massima dell'heap o dello stack.

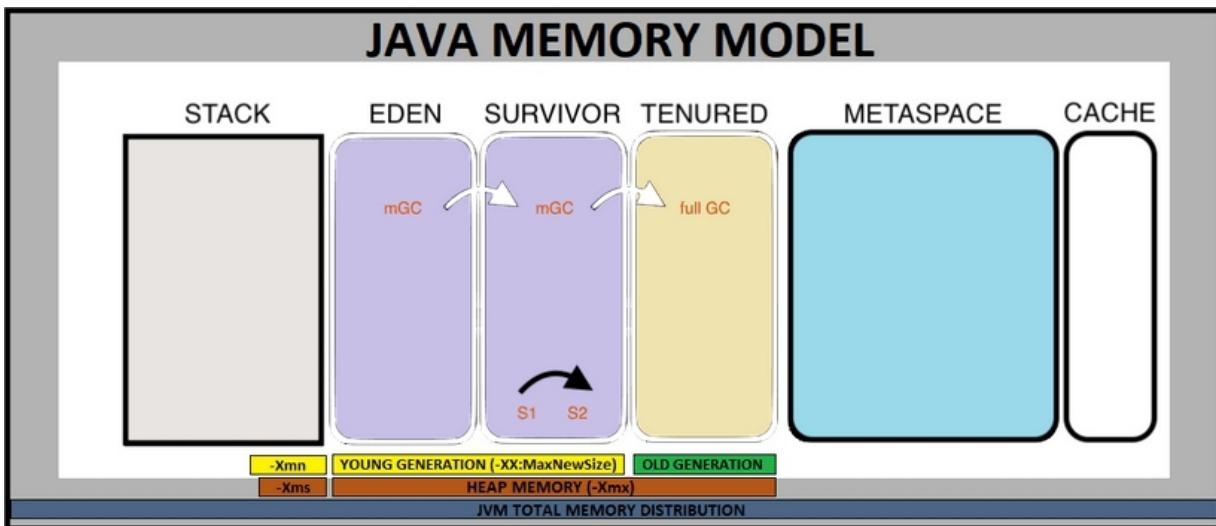
La JVM di Java fornisce diversi parametri di avvio che consentono di configurare le impostazioni di memoria dell'applicazione. I parametri di avvio possono essere specificati mediante l'opzione -X seguita dal nome del parametro e dal valore desiderato.

Ecco alcuni dei parametri di avvio più comuni per la configurazione della memoria nella JVM:

1. **-Xms**: imposta la dimensione iniziale dell'heap, ovvero lo spazio di memoria riservato per l'allocazione degli oggetti. Ad esempio, **-Xms512m** imposta l'heap iniziale a 512 megabyte.
2. **-Xmx**: imposta la dimensione massima dell'heap. Questo parametro definisce il limite massimo di memoria che l'applicazione può utilizzare. Ad esempio, **-Xmx1g** imposta l'heap massimo a 1 gigabyte.
3. **-XX:MaxPermSize**: imposta la dimensione massima dell'area delle costanti, ovvero la memoria riservata per le stringhe, i numeri e le altre costanti utilizzate nel programma.
4. **-Xss**: imposta la dimensione dello stack, ovvero lo spazio di memoria riservato per le variabili locali e le informazioni sullo stato delle chiamate ai metodi.

5. `-XX:NewSize` e `-XX:MaxNewSize`: questi parametri definiscono la dimensione iniziale e massima della memoria riservata per la generazione dei nuovi oggetti, ovvero l'area di memoria dove vengono allocati gli oggetti appena creati.
6. `-XX:SurvivorRatio`: definisce il rapporto tra le dimensioni della generazione giovane e dell'area di sopravvivenza, ovvero l'area di memoria dove vengono trasferiti gli oggetti che hanno superato la prima fase di garbage collection.

La configurazione della memoria nella JVM dipende dalle specifiche esigenze dell'applicazione e dalle risorse disponibili sulla macchina in cui viene eseguita. Una corretta configurazione della memoria può aiutare a migliorare le prestazioni dell'applicazione e prevenire eventuali problemi di esaurimento della memoria o di prestazioni insufficienti.



PermGen (Permanent Generation) è stata un'area di memoria nell'heap di Java utilizzata per la memorizzazione di metadati per la gestione delle classi, come ad esempio i riferimenti alle classi, ai metodi e ai campi.

Tuttavia, la gestione dei metadati nella PermGen poteva causare problemi di memoria come l'esaurimento della memoria, in particolare in applicazioni che caricano dinamicamente molte librerie o usano class loader personalizzati. Inoltre, la PermGen non era soggetta alla garbage collection standard della JVM, ma richiedeva un meccanismo di pulizia separato che poteva essere lento e inefficiente.

Per questo motivo, a partire da Java 8, la PermGen è stata sostituita dalla Metaspace, un'area di memoria dinamica che gestisce i metadati della JVM in modo più efficiente e scalabile, eliminando i problemi di esaurimento della memoria della PermGen e semplificando la gestione della memoria dei metadati.

A differenza dell'area PermGen, il Metaspace non ha un limite fisso predefinito, ma può aumentare dinamicamente in base alle esigenze dell'applicazione.

La gestione del Metaspace è affidata alla JVM, che alloca e dealloca la memoria dinamicamente in base alle esigenze dell'applicazione. Tuttavia, è possibile configurare la dimensione massima del Metaspace tramite il parametro di avvio `-XX:MaxMetaspaceSize`.

INSTALLAZIONE DI JAVA

L'installazione di Java dipende dal sistema operativo in uso. Di seguito sono riportate le istruzioni generali per l'installazione di Java su Windows, macOS e Linux.

Su Windows:

1. Scaricare l'ultima versione di Java dal sito ufficiale di Oracle:
<https://www.oracle.com/java/technologies/javase-downloads.html>
2. Fare doppio clic sul file di installazione scaricato per avviare l'installazione.
3. Seguire le istruzioni guidate dell'installer per completare l'installazione.

Su macOS:

1. Aprire il terminale.
2. Verificare se Java è già installato digitando "java -version" nel terminale. Se Java non è installato, il terminale mostrerà un messaggio di errore.
3. Scaricare l'ultima versione di Java dal sito ufficiale di Oracle:
<https://www.oracle.com/java/technologies/javase-downloads.html>
4. Fare doppio clic sul file di installazione scaricato per avviare l'installazione.
5. Seguire le istruzioni guidate dell'installer per completare l'installazione.

Su Linux:

1. Aprire il terminale.
2. Digitare il comando "sudo apt-get install default-jdk" per installare la versione predefinita di Java disponibile nei

repository di Linux. Se si desidera installare una versione specifica di Java, digitare "sudo apt-get install openjdk-<versione>" al posto di "default-jdk".

3. Digitare "java -version" per verificare se Java è stato installato correttamente e visualizzare la versione installata.

È importante notare che Java viene fornito in diverse edizioni e versioni, a seconda delle esigenze dell'applicazione. Ad esempio, Java Standard Edition (SE) è l'edizione di base di Java utilizzata per lo sviluppo di applicazioni desktop e server, mentre Java Enterprise Edition (EE) è utilizzata per lo sviluppo di applicazioni enterprise complesse. Inoltre, è possibile scegliere tra diverse versioni di Java, come Java 8, Java 11 e Java 17, a seconda delle esigenze dell'applicazione.

JSE e JDK sono due cose diverse ma correlate:

- **JSE (Java Standard Edition)** è l'edizione standard di Java, che contiene le librerie e le API necessarie per lo sviluppo di applicazioni desktop e server. JSE include anche il JRE (Java Runtime Environment), che consente di eseguire le applicazioni Java su una macchina virtuale Java (JVM) senza dover installare il JDK.
- **JDK (Java Development Kit)** è un pacchetto software che include il JSE, insieme a strumenti di sviluppo aggiuntivi come il compilatore Java, il debugger, il profiler e altri strumenti di sviluppo. Il JDK è necessario per sviluppare e compilare applicazioni Java, mentre il JSE è necessario solo per eseguirle.

In sintesi, il JSE è l'edizione standard di Java che contiene le librerie e le API necessarie per lo sviluppo e l'esecuzione di applicazioni Java, mentre il JDK è il pacchetto completo di sviluppo di Java che include il JSE e gli strumenti di sviluppo aggiuntivi. Quindi, se si desidera sviluppare applicazioni Java, è necessario installare il JDK, mentre se si desidera solo eseguire applicazioni Java, è sufficiente installare il JRE (incluso nel JSE).

CONCETTI FONDAMENTALI DELLA PROGRAMMAZIONE

In questo di capitolo parleremo dei fondamenti del linguaggio

Struttura di un programma

In generale, un programma Java ha una struttura composta da alcune parti fondamentali, che possono essere suddivise in tre parti principali:

1. Dichiarazione del pacchetto: la dichiarazione del pacchetto è l'opzione più comune in un programma Java. Questa dichiarazione indica la posizione del file sorgente all'interno della gerarchia del pacchetto. Sebbene questa parte sia opzionale, è buona pratica includerla per aiutare ad organizzare il codice.

```
package nome.pacchetto;
```

2. Importazione delle librerie: Java ha una vasta libreria standard che offre funzioni e classi che possono essere utilizzate in un programma Java. Per accedere alle funzioni di queste librerie, è necessario importarle nel programma.

```
import java.util.ArrayList;
```

3. Definizione della classe principale: Tutti i programmi Java devono contenere almeno una classe che contenga un metodo "main". Questo metodo indica il punto di ingresso del programma e contiene il codice che viene eseguito quando il programma viene avviato.

```
public class NomeClasse {  
    public static void main(String[] args) {  
        // codice del programma  
    }  
}
```

Oltre a queste tre parti principali, un programma Java può includere anche altre classi, metodi, variabili e commenti. Tuttavia, queste tre parti sono essenziali per la creazione di un programma Java funzionante.

È importante notare che ogni riga di codice Java termina con un punto e virgola (;) e che il codice Java è case sensitive, ovvero fa distinzione tra lettere maiuscole e minuscole.

TIPI DI DATO IN JAVA

In Java, i dati vengono manipolati mediante variabili. Le variabili possono contenere dati di diversi tipi, tra cui:

Tipi di dati primitivi: Questi sono i tipi di dati base che Java offre, come int, double, float, boolean, char, short, byte e long. Questi tipi di dati primitivi sono predefiniti dal linguaggio e non hanno bisogno di essere importati.

Tipi di dati di riferimento: Questi sono i tipi di dati definiti dall'utente o le classi Java esistenti. Questi tipi di dati includono stringhe, array, oggetti, classi, interfacce e così via.

Tipi di dati impliciti: Java supporta anche i tipi di dati impliciti, come auto-incremento di int e la promozione di tipi più piccoli a tipi più grandi. Questo significa che, ad esempio, una variabile di tipo byte può essere promossa a un tipo di dato int se necessario.

Quando si dichiara una variabile in Java, è necessario specificare il tipo di dati che la variabile conterrà. Ad esempio, se si desidera dichiarare una variabile intera, si utilizza la sintassi "int nomeVariabile = valoreIniziale".

Esempio:

```
int numero = 5;
double numeroDecimale = 3.14;
boolean flag = true;
String nome = "Mario";
```

In questo esempio, abbiamo dichiarato quattro variabili con tipi di dati diversi. La variabile "numero" è di tipo int e contiene il valore 5, la variabile "numeroDecimale" è di tipo double e contiene il valore 3.14, la variabile "flag" è di tipo boolean e contiene il valore true e la variabile "nome" è di tipo String e contiene la stringa "Mario".

È importante scegliere il tipo di dato giusto per la variabile che si sta dichiarando. In questo modo si può assicurare che il programma funzioni

correttamente e che le variabili contengano il tipo di dato e il valore corretti.

I TIPI PRIMITIVI

In Java, ci sono otto tipi di dato primitivi, che rappresentano i tipi di base dei dati numerici, booleani e caratteri.

Ecco una panoramica dei tipi di dati primitivi in Java:

1. boolean: rappresenta un valore booleano, ovvero vero o falso. Può assumere solo due valori: true o false.
2. byte: rappresenta un numero intero a 8 bit. Il valore minimo è -128 e il valore massimo è 127.
3. short: rappresenta un numero intero a 16 bit. Il valore minimo è -32768 e il valore massimo è 32767.
4. int: rappresenta un numero intero a 32 bit. Il valore minimo è -2^{31} e il valore massimo è $2^{31}-1$.
5. long: rappresenta un numero intero a 64 bit. Il valore minimo è -2^{63} e il valore massimo è $2^{63}-1$.
6. float: rappresenta un numero a virgola mobile a 32 bit, con una precisione di circa 6-7 cifre decimali.
7. double: rappresenta un numero a virgola mobile a 64 bit, con una precisione di circa 15 cifre decimali.
8. char: rappresenta un singolo carattere Unicode a 16 bit.

È importante notare che questi tipi di dati primitivi non sono oggetti, il che significa che non hanno metodi o proprietà. Tuttavia, ci sono classi wrapper corrispondenti per ognuno di questi tipi di dati, come ad esempio Boolean, Byte, Short, Integer, Long, Float, Double e Character, che sono oggetti e forniscono metodi e proprietà utili per lavorare con i dati.

TIPI DI DATO PER RIFERIMENTO

i tipi di dato per riferimento sono classi, interfacce e array. A differenza dei tipi di dato primitivi, i tipi di dato per riferimento sono oggetti che possono avere proprietà e metodi.

Ecco una breve descrizione dei tipi di dato per riferimento in Java:

1. Classi: sono le definizioni delle entità del mondo reale che vogliamo modellare nel nostro programma. Una classe in Java può contenere variabili di istanza (attributi) e metodi (comportamenti) che descrivono l'oggetto. Ad esempio, una classe "Persona" potrebbe contenere attributi come "nome", "cognome" e "data di nascita", insieme a metodi come "calcola l'età" o "stampa le informazioni della persona".
2. Interfacce: sono simili alle classi, ma definiscono solo le firme dei metodi che una classe deve implementare. Una interfaccia può essere considerata come un contratto che definisce cosa una classe deve fare, ma non come la classe dovrebbe farlo. Ad esempio, l'interfaccia "Animale" potrebbe definire il metodo "cammina" che deve essere implementato dalle classi "Cane" e "Gatto".
3. Array: sono strutture dati che consentono di contenere più elementi dello stesso tipo. In Java, gli array sono considerati oggetti e possono contenere elementi di qualsiasi tipo di dato, compresi altri array. Gli elementi di un array sono indicizzati, il che significa che ogni elemento può essere identificato in base alla sua posizione nell'array.

I tipi di dato per riferimento in Java sono importanti perché consentono di creare oggetti complessi e strutture dati sofisticate. Sono fondamentali per lo sviluppo di applicazioni di grandi dimensioni e complessità.

I TIPI DI DATO IMPLICITI

In Java, ci sono due tipi di dati impliciti che vengono utilizzati quando si lavora con espressioni: il widening e l'autoboxing.

1. **Widening:** Il widening è una conversione implicita di un tipo di dato più piccolo in un tipo di dato più grande. Quando si esegue un'operazione con due operandi di tipi di dati diversi, Java eseguirà automaticamente il widening del tipo di dato più piccolo in quello più grande. Ad esempio, se si sommano un intero e un valore float, Java convertirà automaticamente l'intero in un valore float prima di eseguire l'operazione di somma.
2. **Autoboxing:** L'autoboxing è la conversione automatica di un tipo di dato primitivo in un'istanza della classe wrapper corrispondente, e viceversa. Ad esempio, se si utilizza un valore int in un contesto in cui è richiesto un oggetto Integer, Java eseguirà automaticamente l'autoboxing dell'int in un oggetto Integer. Allo stesso modo, se si utilizza un oggetto Integer in un contesto in cui è richiesto un valore int, Java eseguirà automaticamente il "unboxing" dell'oggetto Integer, ovvero la conversione in un valore int primitivo.

È importante notare che il widening e l'autoboxing vengono eseguiti automaticamente dal compilatore Java, ma possono comportare un aumento del tempo di esecuzione e della memoria utilizzata. Pertanto, è importante comprendere quando e come questi tipi di conversione vengono eseguiti, per evitare problemi di performance o errori di compilazione.

un esempio di widening:

```
int numIntero = 10;
float numFloat = 3.14f;
float risultato = numIntero + numFloat;
```

In questo esempio, abbiamo una variabile di tipo int "numIntero" con valore 10, una variabile di tipo float "numFloat" con valore 3.14f e una variabile di tipo float "risultato". Quando eseguiamo l'operazione di somma tra numIntero e numFloat, Java esegue automaticamente il widening dell'intero in un float, in modo che l'operazione di somma possa essere eseguita correttamente. Il valore risultante viene quindi assegnato alla variabile "risultato".

L'autoboxing in Java è il meccanismo automatico che converte un tipo primitivo in un oggetto wrapper corrispondente (ad esempio, int in Integer) quando necessario. Ecco un esempio di autoboxing in Java:

```
int num = 10;
Integer numObj = num; // autoboxing di 'num' in 'numObj'
```

In questo esempio, la variabile num è di tipo primitivo int, mentre numObj è un oggetto Integer. Quando viene assegnato il valore di num a numObj, il valore intero viene automaticamente incapsulato nell'oggetto Integer corrispondente attraverso l'autoboxing. In pratica, si può considerare che il compilatore abbia scritto il codice in questo modo:

```
int num = 10;
Integer numObj = Integer.valueOf(num); // autoboxing esplicito di 'num'
```

In questo caso, la conversione è esplicita e il metodo valueOf viene utilizzato per creare un oggetto Integer a partire dal valore di num. Tuttavia, grazie all'autoboxing, il compilatore esegue questa operazione in modo automatico e trasparente

IL PRIMO PROGRAMMA HELLO WORLD

Ora possiamo creare il primo programma il classico Hello World, un programma che stampa a video la scritta Hello World, il codice del programma è il seguente:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

In questo programma, viene definita una classe chiamata HelloWorld con un metodo main. Quando il programma viene eseguito, il metodo main viene chiamato e il testo "Hello, World!" viene stampato sulla console utilizzando il metodo println della classe System. Per eseguire questo programma, si può utilizzare un'IDE come Eclipse o NetBeans o compilare ed eseguire il file dalla linea di comando usando il seguente comando:

```
javac HelloWorld.java  
java HelloWorld
```

Questo comando compila il file HelloWorld.java e lo esegue. Dovrebbe essere visualizzato il messaggio "Hello, World!" sulla console.

GLI OPERATORI

Gli operatori vengono utilizzati per eseguire operazioni matematiche e logiche su valori e variabili. Ci sono diversi tipi di operatori disponibili in Java, tra cui:

1. **Operatori aritmetici:** questi operatori vengono utilizzati per eseguire operazioni matematiche come l'addizione (+), la sottrazione (-), la moltiplicazione (*), la divisione (/) e il modulo (%). Ad esempio, se si desidera sommare due numeri interi, si può usare l'operatore +:

```
int a = 10;
int b = 5;
int c = a + b; // c contiene il valore 15
```

2. **Operatori di assegnazione:** questi operatori vengono utilizzati per assegnare valori a variabili. L'operatore di assegnazione più comune è "=", ma ci sono anche operatori di assegnazione combinati con gli operatori aritmetici, come "+=" e "-=". Ad esempio, per assegnare il valore 10 alla variabile "a", si può usare l'operatore di assegnazione "=":

```
int a;
a = 10;
```

3. **Operatori di confronto:** questi operatori vengono utilizzati per confrontare due valori e restituire un valore booleano (vero o falso). Gli operatori di confronto più comuni sono "==" (uguale a), "!=" (diverso da), ">" (maggiore di), ">=" (maggiore o uguale a), "<" (minore di) e "<=" (minore o

uguale a). Ad esempio, per verificare se due numeri sono uguali, si può usare l'operatore "==":

```
int a = 10;
int b = 5;
boolean areEqual = (a == b); // areEqual contiene il valore false
```

4. Operatori logici: questi operatori vengono utilizzati per eseguire operazioni logiche su valori booleani. Gli operatori logici più comuni sono "&&" (e logico), "||" (o logico) e "!" (non logico). Ad esempio, per verificare se due condizioni sono entrambe vere, si può usare l'operatore "&&":

```
boolean a = true;
boolean b = false;
boolean result = (a && b); // result contiene il valore false
```

GLI OPERATORI BITWISE

Gli operatori bitwise in Java permettono di manipolare i bit di un valore in modo rapido ed efficiente. Ci sono sei operatori bitwise in Java:

1. AND bit a bit (&): questo operatore esegue l'AND bit a bit tra due valori. Restituisce un valore in cui ogni bit è impostato solo se entrambi i bit corrispondenti nei valori di input sono impostati.
2. OR bit a bit (): questo operatore esegue l'OR bit a bit tra due valori. Restituisce un valore in cui ogni bit è impostato se uno dei bit corrispondenti nei valori di input è impostato.
3. XOR bit a bit (^): questo operatore esegue l'XOR bit a bit tra due valori. Restituisce un valore in cui ogni bit è impostato solo se uno dei bit corrispondenti nei valori di input è impostato, ma non entrambi.
4. NOT (~): questo operatore nega (o inverte) tutti i bit di un valore. Restituisce un valore in cui tutti i bit sono impostati al contrario di quelli nel valore di input.
5. Spostamento a sinistra (<<): questo operatore sposta tutti i bit di un valore verso sinistra di un numero specificato di posizioni. Ogni bit spostato a sinistra viene riempito con uno zero.
6. Spostamento a destra (>>): questo operatore sposta tutti i bit di un valore verso destra di un numero specificato di posizioni. Ogni bit spostato a destra viene riempito con un valore che dipende dal bit più significativo del valore originale (questo è il cosiddetto "bit di segno"). Se il bit di segno è impostato, il bit spostato a destra viene riempito con un valore di 1, altrimenti viene riempito con un valore di 0.

Gli operatori bitwise sono spesso usati in combinazione con le maschere bitwise per manipolare specifici bit in un valore. Spero che questa

spiegazione ti sia stata utile!

*****ebook converter DEMO Watermarks*****

GLI OPERATORI TERNARI

Gli operatori ternari in Java sono un tipo di operatore condizionale che consentono di valutare una condizione e restituire un valore a seconda che la condizione sia vera o falsa. L'operatore ternario ha la seguente sintassi:

```
condizione ? valoreSeVera : valoreSeFalsa;
```

La condizione viene valutata come un'espressione booleana e, se è vera, viene restituito il valore "valoreSeVera". Se la condizione è falsa, viene restituito il valore "valoreSeFalsa".

Ad esempio, supponiamo di avere una variabile "eta" di tipo intero e vogliamo verificare se la persona rappresentata da tale età è maggiorenne. Possiamo utilizzare l'operatore ternario come segue:

```
String messaggio = eta >= 18 ? "La persona è maggiorenne" : "La persona è minorenne";  
System.out.println(messaggio);
```

In questo esempio, la condizione è "eta >= 18", che viene valutata come un'espressione booleana. Se la condizione è vera (cioè se "eta" è maggiore o uguale a 18), viene restituito il valore "La persona è maggiorenne".

Altrimenti, se la condizione è falsa, viene restituito il valore "La persona è minorenne".

L'operatore ternario può essere utilizzato in molte situazioni in cui è necessario effettuare una scelta tra due valori a seconda di una condizione. Tuttavia, è importante usarlo con parsimonia e assicurarsi che il codice risultante sia ancora leggibile e facile da comprendere.

LE CONDIZIONI

In Java, le condizioni sono utilizzate per controllare se una determinata espressione booleana è vera o falsa e decidere di conseguenza quale blocco di codice eseguire. Ci sono diversi tipi di costrutti condizionali in Java:

- **if-else:** è il costrutto condizionale più comune. La sintassi di base è la seguente:

```
if (condizione) {  
    // Blocco di codice da eseguire se la condizione è vera  
} else {  
    // Blocco di codice da eseguire se la condizione è falsa  
}
```

- **nested if:** si tratta di un'istruzione if all'interno di un'altra istruzione if. Questo costrutto può essere utile quando si ha bisogno di controllare più condizioni in sequenza. La sintassi è la seguente:

```
if (condizione1) {  
    if (condizione2) {  
        // Blocco di codice da eseguire se entrambe le condizioni sono vere  
    } else {  
        // Blocco di codice da eseguire se la seconda condizione è falsa  
    }  
} else {  
    // Blocco di codice da eseguire se la prima condizione è falsa  
}
```

- **switch-case:** è un costrutto che consente di selezionare un blocco di codice da eseguire in base al valore di una variabile. La sintassi è la seguente:

```
switch (variabile) {
    case valore1:
        // Blocco di codice da eseguire se la variabile è uguale a valore1
        break;
    case valore2:
        // Blocco di codice da eseguire se la variabile è uguale a valore2
        break;
    default:
        // Blocco di codice da eseguire se la variabile non corrisponde a nessuno
}
```

- **ternary operator:** come ho spiegato precedentemente, l'operatore ternario consente di valutare una condizione e restituire un valore a seconda che la condizione sia vera o falsa. La sintassi è la seguente:

```
valoreDaAssegnare = condizione ? valoreSeVera : valoreSeFalsa;
```

In sintesi, i costrutti condizionali sono utilizzati per eseguire determinati blocchi di codice solo se una condizione specifica è vera. In questo modo, è possibile scrivere programmi più flessibili e reattivi che si adattano alle varie situazioni che possono verificarsi durante l'esecuzione del programma.

I CICLI

un ciclo è una struttura di controllo che permette di eseguire ripetutamente un blocco di codice fino a quando una certa condizione viene soddisfatta.

Ci sono tre tipi di cicli in Java: il ciclo **while**, il ciclo **do-while** e il ciclo **for**.

Il ciclo while: Il ciclo while esegue il blocco di codice finché una condizione specificata risulta vera. La sintassi è la seguente:

```
while (condizione) {  
    // codice da eseguire finché la condizione è vera  
}
```

Esempio di utilizzo:

```
int i = 0;  
while (i < 10) {  
    System.out.println(i);  
    i++;  
}
```

Il ciclo do-while: Il ciclo do-while è simile al ciclo while, ma il blocco di codice viene eseguito almeno una volta prima di controllare la condizione. La sintassi è la seguente:

```
do {  
    // codice da eseguire almeno una volta  
} while (condizione);
```

Esempio di utilizzo:

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 10);
```

Il ciclo for: Il ciclo for è utilizzato quando si conosce il numero di volte che il blocco di codice deve essere eseguito. La sintassi è la seguente:

```
for (inizializzazione; condizione; incremento) {
    // codice da eseguire finché la condizione è vera
}
```

Esempio di utilizzo:

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

In questo esempio, il ciclo for esegue il blocco di codice 10 volte, incrementando la variabile i ad ogni iterazione.

CONTROLLO DEL FLUSSO DI UN CICLO

le istruzioni **break** e **continue** sono utilizzate per controllare il flusso di esecuzione all'interno di un ciclo (for, while, do-while o switch-case).

L'istruzione **break** viene utilizzata per interrompere l'esecuzione di un ciclo o di uno switch-case. Quando viene eseguita, l'istruzione **break** fa uscire immediatamente dal ciclo o dallo switch-case, senza eseguire il resto del codice all'interno del blocco. L'istruzione **break** viene spesso utilizzata in combinazione con un'istruzione condizionale all'interno di un ciclo per controllare quando interrompere il ciclo. Ad esempio:

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) {  
        break; // interrompe il ciclo quando i = 5  
    }  
    System.out.println(i);  
}
```

L'istruzione **continue**, invece, viene utilizzata per saltare una singola iterazione di un ciclo. Quando viene eseguita, l'istruzione **continue** fa saltare la singola iterazione in corso e passare direttamente alla successiva, senza eseguire il resto del codice all'interno del blocco per quella iterazione.

L'istruzione **continue** viene spesso utilizzata in combinazione con un'istruzione condizionale all'interno di un ciclo per controllare quando saltare l'iterazione. Ad esempio:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // salta l'iterazione quando i = 5
    }
    System.out.println(i);
}
```

In questo esempio, l'istruzione continue viene eseguita quando i è uguale a 5. Questo fa saltare l'iterazione corrente e passare alla successiva, senza eseguire la riga System.out.println(i) per i=5.

In sintesi, le istruzioni break e continue sono utilizzate per controllare il flusso di esecuzione all'interno di un ciclo o di uno switch-case. L'istruzione break fa uscire immediatamente dal ciclo o dallo switch-case, mentre l'istruzione continue fa saltare una singola iterazione del ciclo

GESTIONE DELL'INPUT E OUTPUT DA CONSOLE

Per poter interagire con i nostri primi programmi sarà utile imparare a gestire l'input e l'output della console, per poter utilizzare queste funzioni si utilizzano le classi System.in, System.out e System.err della libreria standard.

Per la lettura dell'input da tastiera, si utilizza l'oggetto Scanner che permette di leggere i dati inseriti dall'utente. Ecco un esempio:

```
import java.util.Scanner;

public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Inserisci un numero intero: ");
        int numero = scanner.nextInt();
        System.out.println("Il numero inserito è: " + numero);
    }
}
```

In questo esempio, si crea un'istanza della classe Scanner passando come parametro l'oggetto System.in, ovvero lo stream di input standard. Successivamente, si utilizza il metodo nextInt() per leggere l'input dell'utente, che viene salvato nella variabile numero.

Per la scrittura dell'output sulla console, si utilizza il metodo println() dell'oggetto System.out. Ecco un esempio:

```
public class ConsoleOutputExample {  
    public static void main(String[] args) {  
        System.out.println("Ciao mondo!");  
    }  
}
```

In questo esempio, viene utilizzato il metodo `println()` per scrivere la stringa "Ciao mondo!" sulla console.

Per gestire gli errori, si utilizza l'oggetto `System.err`, che funziona in modo simile a `System.out`. Ecco un esempio:

```
public class ConsoleErrorExample {  
    public static void main(String[] args) {  
        System.err.println("Errore: divisione per zero!");  
    }  
}
```

In questo esempio, viene utilizzato il metodo `println()` dell'oggetto `System.err` per scrivere il messaggio di errore "Errore: divisione per zero!" sulla console.

Si può anche utilizzare la classe `BufferedReader` per la lettura dell'input, e la classe `PrintWriter` per la scrittura dell'output e degli errori, ma queste classi richiedono un po' più di codice per la loro configurazione.

IL METODO MAIN

il metodo main è il punto di ingresso per l'esecuzione di un programma. Esso è il primo metodo che viene eseguito quando si avvia un'applicazione Java e ha la seguente firma:

```
public static void main(String[] args) {  
    // corpo del metodo  
}
```

La parola chiave public indica che il metodo può essere chiamato da qualsiasi classe, mentre static indica che il metodo appartiene alla classe stessa e non all'istanza della classe. Il tipo di ritorno è void, che significa che il metodo non restituisce alcun valore.

Il parametro args è un array di stringhe che rappresenta gli argomenti passati al programma dalla riga di comando. Quando si avvia un programma Java dalla riga di comando, si possono specificare uno o più argomenti separati da spazi.

ESERCIZIO 1

Scrivere un programma Java che dando n numeri in input ne produce la media, il programma si interromperà dando come input il comando q, visualizzando il risultato.

Soluzione:

```
1 package javabase.capitolo2;
2
3 import java.util.Scanner;
4
5 public class MediaNumeri {
6     public static void main(String[] args) {
7         Scanner input = new Scanner(System.in);
8         double somma = 0;
9         int contatore = 0;
10
11         while (true) {
12             System.out.print("Inserisci un numero (q per uscire): ");
13             String inputString = input.nextLine();
14
15             // Verifica se l'utente ha inserito il carattere "q"
16             if (inputString.equals("q")) {
17                 break; // esce dal ciclo while
18             }
19
20             // Converte la stringa in un numero double
21             double numero = Double.parseDouble(inputString);
22
23             // Aggiorna la somma e il contatore
24             somma += numero;
25             contatore++;
26         }
27
28         // Calcola la media
29         double media = somma / contatore;
30
31         // Stampa il risultato
32         System.out.println("La media dei numeri inseriti è: " + media);
33     }
34 }
35 }
```

Il programma utilizza un ciclo while per continuare a chiedere all'utente di inserire numeri finché non viene inserito il carattere "q". All'interno del ciclo, la stringa inserita dall'utente viene convertita in un numero double e la somma e il contatore vengono aggiornati di conseguenza.

Una volta che l'utente ha inserito il carattere "q", il ciclo si interrompe e il programma calcola la media dei numeri inseriti dividendo la somma per il contatore. Infine, il programma stampa il risultato della media dei numeri inseriti.

Nota che il programma è in grado di gestire anche il caso in cui l'utente inserisca una stringa non valida che non può essere convertita in un numero double, ad esempio "ciao". In questo caso, il programma lancerà un'eccezione di tipo **NumberFormatException**, che potrebbe essere gestita in modo appropriato in base alle esigenze specifiche del programma, ma ne parleremo più in dettaglio nel capitolo delle eccezioni

ESERCIZIO 2

scrivere un programma java che prende in input i voti di tre materie Italiano, Matematica e Geografia ne calcoli la media e produca come output la scritta promosso se la media è maggiore del 6 o bocciato se è inferiore al 6.

Soluzione:

```
1 package javabase.capitolo2;
2
3 import java.util.Scanner;
4
5 public class CalcolaMediaVoti {
6     public static void main(String[] args) {
7         Scanner input = new Scanner(System.in);
8
9         // Chiede all'utente di inserire i voti per le tre materie
10        System.out.print("Inserisci il voto in Italiano: ");
11        double votoItaliano = input.nextDouble();
12
13        System.out.print("Inserisci il voto in Matematica: ");
14        double votoMatematica = input.nextDouble();
15
16        System.out.print("Inserisci il voto in Geografia: ");
17        double votoGeografia = input.nextDouble();
18
19        // Calcola la media dei voti
20        double media = (votoItaliano + votoMatematica + votoGeografia) / 3;
21
22        // Decide se lo studente è promosso o bocciato in base alla media dei voti
23        if (media >= 6) {
24            System.out.println("Promosso! La tua media dei voti è " + media);
25        } else {
26            System.out.println("Bocciato! La tua media dei voti è " + media);
27        }
28    }
29}
30}
```

Il programma chiede all'utente di inserire i voti per le tre materie utilizzando l'oggetto Scanner, quindi calcola la media dei voti. Successivamente, utilizzando un'istruzione condizionale if-else, il programma decide se lo studente è promosso o bocciato in base alla media

dei voti e stampa il risultato. Se la media è maggiore o uguale a 6, lo studente è promosso, altrimenti è bocciato.

ESERCIZIO 3

Scrivere un programma Java che chiede all'utente di inserire un numero intero compreso tra 1 e 7, che rappresenta un giorno della settimana, e stampa il nome del giorno corrispondente utilizzando l'istruzione switch. Se l'utente inserisce un numero non compreso tra 1 e 7, il programma deve stampare un messaggio di errore.

```
5 public class GiornoSettimana {
6     public static void main(String[] args) {
7         Scanner input = new Scanner(System.in);
8         System.out.print("Inserisci un numero da 1 a 7: ");
9         int numero = input.nextInt();
10
11         String nomeGiorno;
12         switch(numero) {
13             case 1:
14                 nomeGiorno = "Lunedì";
15                 break;
16             case 2:
17                 nomeGiorno = "Martedì";
18                 break;
19             case 3:
20                 nomeGiorno = "Mercoledì";
21                 break;
22             case 4:
23                 nomeGiorno = "Giovedì";
24                 break;
25             case 5:
26                 nomeGiorno = "Venerdì";
27                 break;
28             case 6:
29                 nomeGiorno = "Sabato";
30                 break;
31             case 7:
32                 nomeGiorno = "Domenica";
33                 break;
34             default:
35                 System.out.println("Numero non valido! Inserisci un numero da 1 a 7.");
36                 return;
37         }
38
39         System.out.println("Il giorno corrispondente è " + nomeGiorno + ".");
40     }
41 }
```

Il programma utilizza l'istruzione switch per assegnare il nome del giorno corrispondente al numero inserito dall'utente. Se il numero inserito

dall'utente non è compreso tra 1 e 7, il programma stampa un messaggio di errore. L'istruzione return viene utilizzata per terminare il programma se viene inserito un numero non valido.

ESERCIZIO 4

Scrivere un programma Java che chiede all'utente di inserire due numeri interi e li confronta utilizzando gli operatori bitwise. In particolare, il programma deve:

1. Stampa i due numeri inseriti dall'utente.
2. Calcolare la somma dei due numeri utilizzando l'operatore | bitwise.
3. Calcolare la differenza dei due numeri utilizzando l'operatore & bitwise.
4. Calcolare il prodotto dei due numeri utilizzando l'operatore ^ bitwise.
5. Stampa i risultati ottenuti.

Esempio di esecuzione del programma:

```
Inserisci il primo numero: 5
Inserisci il secondo numero: 9

Numeri inseriti: 5 e 9
Somma bitwise: 13
Differenza bitwise: 1
Prodotto bitwise: 12
```

Soluzione:

```
1 package javabase.capitolo2;
2
3 import java.util.Scanner;
4
5 public class BitwiseOperator {
6     public static void main(String[] args) {
7         Scanner input = new Scanner(System.in);
8         System.out.print("Inserisci il primo numero: ");
9         int num1 = input.nextInt();
10        System.out.print("Inserisci il secondo numero: ");
11        int num2 = input.nextInt();
12
13        System.out.println("Numeri inseriti: " + num1 + " e " + num2);
14
15        int somma = num1 | num2;
16        int differenza = num1 & num2;
17        int prodotto = num1 ^ num2;
18
19        System.out.println("Somma bitwise: " + somma);
20        System.out.println("Differenza bitwise: " + differenza);
21        System.out.println("Prodotto bitwise: " + prodotto);
22    }
23 }
```

Il programma utilizza gli operatori bitwise |, &, e ^ per calcolare la somma, la differenza e il prodotto dei due numeri inseriti dall'utente. I risultati vengono quindi stampati a video.

FUNZIONI

Le Funzioni

Come abbiamo visto una classe java è formata da metodi o funzioni e attributi, ma quali vantaggi apportano le funzioni nel mio codice?

Ci sono diversi vantaggi nell'utilizzare le funzioni in Java:

1. **Modularità:** definendo funzioni ben strutturate e coese, è possibile scrivere codice più modulare e più facile da leggere, mantenere e modificare. La suddivisione del codice in funzioni più piccole e specializzate permette di concentrarsi su un particolare aspetto del problema che si sta risolvendo e di riuscire a creare un codice più strutturato e organizzato.
2. **Riusabilità:** le funzioni possono essere scritte una volta e poi riutilizzate in diverse parti del codice. Questo significa che non è necessario scrivere codice ripetitivo ogni volta che bisogna eseguire una determinata operazione. Invece, basta richiamare la funzione e il lavoro è fatto.
3. **Testabilità:** le funzioni modulari sono più facili da testare in quanto si può testare ogni funzione separatamente, verificando che ognuna svolga correttamente il suo compito. Inoltre, le funzioni ben strutturate sono anche più facili da testare perché ci si può concentrare solo sulla logica interna della funzione, senza dover preoccuparsi di tutto il contesto dell'applicazione.
4. **Leggibilità:** le funzioni ben strutturate possono migliorare la leggibilità del codice perché possono essere usate per nascondere dettagli complessi di implementazione. Ad esempio, si può scrivere una funzione che esegue un'operazione complessa, come la gestione di una connessione a un database, e chiamarla semplicemente con un nome significativo senza dover preoccuparsi dei dettagli di come viene gestita la connessione.

Scomporre il codice in più funzioni può essere un processo creativo che richiede un po' di pratica e sperimentazione. Ecco alcuni suggerimenti su come capire dove dividere il codice in funzioni più piccole:

1. Cerca le parti di codice che svolgono compiti specifici: identifica le parti del codice che svolgono compiti specifici e distinti, come ad esempio la gestione di input/output, il calcolo di una media, l'ordinamento di una lista, ecc.
2. Trova parti di codice ripetitive: se hai parti di codice che vengono ripetute più volte, puoi considerare la possibilità di scrivere una funzione per gestirle in modo più efficiente.
3. Rileva parti di codice complesse o difficili da leggere: se ci sono parti di codice che sono particolarmente complesse o difficili da leggere, puoi considerare la possibilità di separarle in una funzione a sé stante, in modo da renderle più leggibili e più facili da comprendere.
4. Usa nomi significativi per le funzioni: assicurati di dare nomi significativi alle funzioni che crei, in modo da rendere chiaro il loro scopo e facilitare la comprensione del codice.

In generale, l'obiettivo è quello di dividere il codice in funzioni più piccole e specializzate, in modo da rendere il codice più modulare, riutilizzabile, testabile e leggibile. Inoltre, una buona pratica è quella di mantenere le funzioni relativamente brevi, in modo che siano più facili da comprendere e gestire.

Le funzioni sono definite come metodi all'interno delle classi. Per definire una funzione in Java, si deve specificare il nome del metodo, il tipo di valore di ritorno (se esiste) e i parametri necessari. Il tipo di valore di ritorno indica il tipo di dato che la funzione restituisce quando viene chiamata.

Ad esempio, ecco come si definisce una funzione che prende in input due numeri interi e restituisce la loro somma:

```
public class Calcolatrice {  
    public static int somma(int num1, int num2) {  
        int risultato = num1 + num2;  
        return risultato;  
    }  
}
```

in questo esempio, abbiamo definito una classe Calcolatrice e un metodo somma all'interno della classe. Il metodo somma prende in input due parametri di tipo int chiamati num1 e num2. Il corpo del metodo calcola la somma dei due numeri e la restituisce utilizzando l'istruzione return.

Per chiamare il metodo somma dall'esterno della classe, dobbiamo prima creare un'istanza della classe e poi utilizzare la sintassi nome_istanza.nome_methodo() per chiamare il metodo. Ad esempio:

```
public class EsempioChiamataMetodo {  
    public static void main(String[] args) {  
        Calcolatrice calcolatrice = new Calcolatrice();  
        int risultato = calcolatrice.somma(5, 7);  
        System.out.println("La somma di 5 e 7 è " + risultato);  
    }  
}
```

In questo esempio, abbiamo creato un'istanza della classe Calcolatrice chiamata calcolatrice e poi abbiamo chiamato il metodo somma utilizzando la sintassi calcolatrice.somma(5, 7). Il valore restituito dal metodo viene memorizzato nella variabile risultato e viene quindi stampato a video utilizzando l'istruzione System.out.println().

I PARAMETRI

I parametri formali si riferiscono alle variabili definite all'interno della dichiarazione di una funzione o metodo, mentre i parametri attuali si riferiscono ai valori passati alla funzione o al metodo al momento della chiamata.

I parametri formali vengono specificati durante la definizione di una funzione o metodo. Essi descrivono il tipo e il nome della variabile che verrà utilizzata all'interno della funzione o del metodo per accedere al valore passato come parametro. Ad esempio, in un metodo che calcola la somma di due numeri, i parametri formali potrebbero essere definiti come "int a" e "int b".

I parametri attuali sono i valori concreti passati alla funzione o al metodo al momento della chiamata. Essi devono corrispondere in tipo e ordine ai parametri formali specificati nella definizione della funzione o del metodo. Ad esempio, se la funzione che calcola la somma di due numeri ha i parametri formali "int a" e "int b", i parametri attuali potrebbero essere "3" e "4", passati nella chiamata come "sum(3, 4)".

Durante l'esecuzione del programma, i parametri attuali vengono utilizzati per assegnare i valori ai parametri formali all'interno della funzione o del metodo. I parametri formali vengono utilizzati all'interno della funzione o del metodo per elaborare i dati passati come parametri attuali.

In sostanza, i parametri formali sono le variabili che definiamo all'interno della funzione o del metodo per utilizzarle come input, mentre i parametri attuali sono i valori concreti che passiamo alla funzione o al metodo al momento della chiamata, in modo che questi vengano utilizzati all'interno della funzione o del metodo per eseguire le operazioni necessarie.

i parametri di un metodo possono essere passati per valore o per riferimento. Questo può influenzare il comportamento del programma quando si lavora con oggetti e array.

Nel passaggio per valore, viene passata una copia del valore del parametro al metodo. Questo significa che qualsiasi modifica apportata al parametro

all'interno del metodo non avrà alcun effetto sul valore originale del parametro all'esterno del metodo. Ad esempio:

```
public class PassaggioPerValore {
    public static void incrementa(int x) {
        x = x + 1;
    }

    public static void main(String[] args) {
        int valore = 5;
        incrementa(valore);
        System.out.println("Il valore è " + valore);
    }
}
```

In questo esempio, abbiamo definito un metodo chiamato incrementa che prende in input un parametro x di tipo int e lo incrementa di 1. Nel metodo main, abbiamo creato una variabile valore e l'abbiamo inizializzata a 5. Abbiamo quindi chiamato il metodo incrementa passando la variabile valore come parametro. Tuttavia, quando viene stampato il valore di valore a video, il valore è ancora 5, perché la variabile x all'interno del metodo incrementa è una copia del valore di valore.

Nel passaggio per riferimento, invece, viene passato un riferimento all'oggetto o all'array che rappresenta il parametro. Questo significa che le modifiche apportate all'oggetto o all'array all'interno del metodo si riflettono anche sull'oggetto o sull'array originale all'esterno del metodo. Ad esempio:

```
public class PassaggioPerRiferimento {
    public static void incrementa(int[] array) {
        for (int i = 0; i < array.length; i++) {
            array[i] = array[i] + 1;
        }
    }

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};
        incrementa(array);
        for (int i = 0; i < array.length; i++) {
            System.out.println("Elemento " + i + " = " + array[i]);
        }
    }
}
```

In questo esempio, abbiamo definito un metodo chiamato incrementa che prende in input un parametro array di tipo int[] e incrementa ogni elemento dell'array di 1. Nel metodo main, abbiamo creato un array array e l'abbiamo inizializzato con i valori 1, 2, 3, 4 e 5. Abbiamo quindi chiamato il metodo incrementa passando l'array come parametro. Quando viene stampato ogni elemento dell'array a video, il valore di ciascun elemento è stato incrementato di 1, perché l'array originale è stato modificato all'interno del metodo.

In sintesi, il passaggio per valore copia il valore del parametro, mentre il passaggio per riferimento copia il riferimento all'oggetto o all'array che rappresenta il parametro. Quando si lavora con oggetti e array, è importante capire il passaggio per riferimento, perché le modifiche apportate, ma questo lo vedremo nel capitolo degli array.

LE FUNZIONI RICORSIVE

Le funzioni ricorsive sono funzioni che si richiamano da sole all'interno del loro corpo. In altre parole, una funzione ricorsiva è una funzione che chiama se stessa come parte della sua esecuzione.

Quando una funzione viene chiamata in modo ricorsivo, essa si divide in una serie di sotto problemi più piccoli, ciascuno dei quali viene risolto tramite una chiamata ricorsiva. Questi sotto problemi continuano a dividersi in ulteriori sotto problemi fino a quando non si arriva a un caso base, ovvero un caso che può essere risolto direttamente senza ulteriori chiamate ricorsive.

Ad esempio, una funzione ricorsiva potrebbe essere utilizzata per calcolare il fattoriale di un numero. Il fattoriale di un numero è il prodotto di tutti i numeri interi positivi da 1 a quel numero. Quindi, il fattoriale di 5 (5!) sarebbe $1 \times 2 \times 3 \times 4 \times 5$, ovvero 120.

Ecco un esempio di una funzione ricorsiva per calcolare il fattoriale di un numero:

```
public static int factorial(int n) {  
    if (n == 1) { // caso base  
        return 1;  
    } else {  
        return n * factorial(n - 1); // chiamata ricorsiva  
    }  
}
```

In questo esempio, la funzione factorial() si richiama in modo ricorsivo fino a quando non arriva al caso base, che è quando il valore di n è uguale a 1. A questo punto, la funzione restituisce il valore 1 e l'esecuzione ritorna alla

chiamata precedente. La funzione chiamante utilizza quindi il valore restituito dalla chiamata ricorsiva per calcolare il valore finale.

Le funzioni ricorsive possono essere utili per risolvere problemi che hanno una struttura ripetitiva o ricorsiva, come il calcolo di fattoriali o la ricerca in profondità di una struttura dati come un albero. Tuttavia, è importante prestare attenzione alla gestione della memoria e dell'allocazione, in quanto le chiamate ricorsive possono portare a un consumo elevato di memoria e a potenziali problemi di stack overflow se non gestite correttamente.

Individuare quando creare una funzione ricorsiva può essere un compito impegnativo, ma ci sono alcune linee guida che puoi seguire per aiutarti a decidere.

In generale, le funzioni ricorsive sono utilizzate quando un problema può essere suddiviso in sotto-problemi simili o identici, ciascuno dei quali può essere risolto utilizzando la stessa procedura. Ci sono alcuni casi comuni in cui le funzioni ricorsive possono essere utili:

- Strutture dati ricorsive come alberi, grafi e liste concatenate
- Problemi matematici che possono essere definiti in modo ricorsivo, come il calcolo di fattoriali o la successione di Fibonacci
- Algoritmi di ricerca o di ordinamento che richiedono la suddivisione di un problema in sotto-problemi

In generale, se un problema può essere risolto attraverso un approccio "divide-et-impera", ovvero la suddivisione del problema in sotto problemi più piccoli, allora potrebbe essere utile considerare l'uso di una funzione ricorsiva.

Inoltre, se si sta scrivendo codice che utilizza ripetitivamente un blocco di istruzioni simili, allora questo potrebbe essere un'indicazione che il codice potrebbe essere estratto in una funzione ricorsiva.

Infine, la decisione di utilizzare una funzione ricorsiva dipende anche dalle tue preferenze personali e dal tuo stile di programmazione. Alcuni sviluppatori preferiscono scrivere codice iterativo, mentre altri preferiscono utilizzare funzioni ricorsive. In generale, è importante scegliere l'approccio che ti sembra più naturale e che porta al codice più chiaro e manutenibile.

Le funzioni ricorsive hanno diversi vantaggi:

1. Chiarità del codice: Le funzioni ricorsive possono rendere il codice più chiaro e leggibile. Spesso, infatti, i problemi che si prestano ad una soluzione ricorsiva possono essere espressi in modo molto più chiaro e conciso utilizzando una funzione ricorsiva piuttosto che una soluzione iterativa.
2. Modularità del codice: Le funzioni ricorsive possono aumentare la modularità del codice, ovvero la sua capacità di essere diviso in parti indipendenti e facili da gestire. Una funzione ricorsiva può essere scritta in modo da risolvere un singolo sotto-problema, e poi richiamata in modo ricorsivo per risolvere il problema completo.
3. Efficienza: In alcuni casi, le funzioni ricorsive possono essere più efficienti di una soluzione iterativa. Ad esempio, quando si lavora con strutture dati come alberi o grafi, le funzioni ricorsive possono essere molto più efficienti delle soluzioni iterative.
4. Facilità di debugging: Le funzioni ricorsive possono essere più facili da debuggere rispetto alle soluzioni iterative. Quando si utilizza una funzione ricorsiva, è possibile utilizzare un debugger per esaminare il valore di una variabile in un certo punto della chiamata ricorsiva. Questo può essere molto utile per individuare e correggere gli errori nel codice.

In generale, le funzioni ricorsive possono essere molto utili per la risoluzione di determinati problemi, ma è importante ricordare che devono essere utilizzate con cautela, in quanto possono essere molto costose in termini di memoria e di tempo di esecuzione. Inoltre, quando si utilizzano funzioni ricorsive, è importante prestare attenzione al caso base e al caso ricorsivo, in modo da evitare cicli infiniti e crash del programma.

Le funzioni ricorsive possono anche presentare alcuni problemi:

1. Consumo di memoria: Ogni volta che una funzione ricorsiva viene chiamata, viene allocata una nuova area di memoria per i parametri e le variabili locali. Se la funzione viene chiamata molte volte o se il problema da risolvere è molto grande, il consumo di memoria può diventare un problema.

2. Stack overflow: Se la funzione ricorsiva viene chiamata un numero elevato di volte, il sistema operativo può esaurire lo stack di chiamate e causare un errore di "stack overflow". Questo può causare il blocco o il crash del programma.
3. Complessità del codice: In alcuni casi, una soluzione ricorsiva può essere più complessa e meno intuitiva di una soluzione iterativa. Questo può rendere il codice più difficile da leggere, mantenere e debuggare.
4. Performance: In alcuni casi, una soluzione iterativa può essere più efficiente di una soluzione ricorsiva. Ciò può essere dovuto al fatto che l'overhead delle chiamate ricorsive può influire sulla performance del programma.

In generale, è importante considerare attentamente l'utilizzo delle funzioni ricorsive e valutare i loro vantaggi e svantaggi rispetto alle soluzioni iterative. Quando si utilizzano le funzioni ricorsive, è importante prestare attenzione al caso base e al caso ricorsivo, per evitare cicli infiniti e problemi di performance.

ESERCIZIO 5

Scrivere un programma Java che, presi in input due numeri interi, restituisca il massimo comune divisore tra di essi.

Il massimo comune divisore (MCD) è il numero più grande che divide entrambi i numeri senza lasciare resto. Ad esempio, l'MCD tra 12 e 18 è 6, poiché 6 è il più grande numero che divide sia 12 che 18.

Per trovare l'MCD, si può utilizzare l'algoritmo di Euclide, che prevede il calcolo del resto della divisione tra i due numeri e l'uso di questi resti per continuare a dividere i numeri fino a quando uno dei resti è zero. Quando ciò accade, il numero più grande diviso per il resto diverso da zero è l'MCD.

Soluzione:

```
1 package javabase.capitolo3;
2
3 import java.util.Scanner;
4
5 public class MCD {
6     public static void main(String[] args) {
7         Scanner input = new Scanner(System.in);
8         int num1, num2;
9
10        System.out.print("Inserisci il primo numero: ");
11        num1 = input.nextInt();
12
13        System.out.print("Inserisci il secondo numero: ");
14        num2 = input.nextInt();
15
16        int mcd = calcolaMCD(num1, num2);
17
18        System.out.println("Il massimo comune divisore tra " + num1 + " e " + num2 + " è: " + mcd);
19    }
20
21    public static int calcolaMCD(int a, int b) {
22        if (b == 0) {
23            return a;
24        } else {
25            int resto = a % b;
26            return calcolaMCD(b, resto);
27        }
28    }
29}
```

In questo programma, la funzione calcolaMCD è la funzione che effettivamente calcola l'MCD tra i due numeri. Prende in input due numeri

interi a e b e restituisce l'MCD.

La funzione utilizza l'algoritmo di Euclide per calcolare l'MCD. Se il secondo numero è zero, allora l'MCD è il primo numero. Altrimenti, viene calcolato il resto della divisione tra i due numeri, e la funzione viene richiamata nuovamente passando come primo parametro il secondo numero e come secondo parametro il resto appena calcolato.

La funzione viene richiamata in main passando i due numeri inseriti dall'utente. L'MCD calcolato viene poi stampato a video.

ESERCIZIO 6

Scrivere una funzione ricorsiva che calcola il fattoriale di un numero intero positivo.

Il fattoriale di un numero è definito come il prodotto di tutti i numeri interi positivi da 1 a quel numero, ad esempio il fattoriale di 5 è $5 \times 4 \times 3 \times 2 \times 1 = 120$.

Soluzione:

```
1 package javabase.capitolo3;
2
3 public class Fattoriale {
4
5     public static int calcolaFattoriale(int n) {
6         if (n == 1) {
7             return 1;
8         } else {
9             return n * calcolaFattoriale(n - 1);
10        }
11    }
12
13    public static void main(String[] args) {
14        int num = 5;
15        int fatt = calcolaFattoriale(num);
16        System.out.println("Il fattoriale di " + num + " è " + fatt);
17    }
18}
19
```

In questo esempio, la funzione calcolaFattoriale è una funzione ricorsiva che riceve in input un intero n e restituisce il fattoriale di n.

La funzione base della ricorsione è quando n è uguale a 1, in questo caso restituiamo il valore 1. Altrimenti, calcoliamo il fattoriale di n-1 e lo moltiplichiamo per n, ottenendo così il fattoriale di n.

Nel metodo main, viene creato un oggetto Fattoriale e viene invocata la funzione calcolaFattoriale passando il valore di num. Infine, viene visualizzato il risultato del fattoriale calcolato.

ARRAY E COLLEZIONI

In Java, gli array e le collezioni sono strutture dati fondamentali per gestire grandi quantità di dati in modo efficiente.

Gli array sono strutture dati che consentono di memorizzare una sequenza di valori dello stesso tipo. In Java, gli array possono essere di tipo primitivo (come int, double, boolean, etc.) o di tipo oggetto (come String, Object, etc.). Gli array in Java hanno una dimensione fissa, che deve essere specificata al momento della creazione dell'array. La dimensione dell'array può essere modificata solo creando un nuovo array con una dimensione diversa e copiando i valori dell'array originale.

Le collezioni, d'altra parte, sono strutture dati più flessibili rispetto agli array, che permettono di gestire grandi quantità di dati in modo dinamico. In Java, le collezioni sono implementate tramite una serie di interfacce che definiscono i comportamenti comuni a tutti i tipi di collezioni (come ad esempio l'aggiunta, la rimozione, la ricerca, etc.). Le implementazioni di queste interfacce includono diverse strutture dati, come ad esempio le liste (come ArrayList, LinkedList), gli insiemi (come HashSet, TreeSet) e le mappe (come HashMap, TreeMap).

Le collezioni possono contenere oggetti di qualsiasi tipo (tranne i tipi primitivi, che devono essere incapsulati in oggetti wrapper come Integer, Double, etc.), e possono essere di dimensione dinamica, cioè la dimensione della collezione può essere modificata in modo flessibile a seconda delle necessità del programma. Inoltre, le collezioni offrono molte funzionalità avanzate, come ad esempio la possibilità di ordinare gli elementi, eseguire operazioni su tutti gli elementi della collezione (come ad esempio la mappa di una funzione), filtrare gli elementi in base a determinati criteri, e molto altro ancora.

In sintesi, gli array sono utilizzati principalmente per memorizzare sequenze di valori dello stesso tipo, mentre le collezioni sono utilizzate per gestire grandi quantità di dati in modo dinamico e flessibile, e offrono funzionalità avanzate per lavorare con gli elementi contenuti nella collezione.

Ci sono diversi vantaggi nell'utilizzo degli array in Java:

Organizzazione dei dati: gli array possono organizzare grandi quantità di dati in modo ordinato e accessibile attraverso un singolo nome di variabile.

Efficienza: gli array forniscono un accesso rapido ai dati perché tutti gli elementi sono memorizzati in un'unica area di memoria contigua e possono essere acceduti tramite l'indice.

Accesso casuale: gli elementi dell'array possono essere acceduti casualmente, ovvero non è necessario scorrere l'intero array per trovare un elemento specifico.

Sintassi concisa: gli array possono semplificare il codice quando si lavora con grandi quantità di dati. Ad esempio, se si devono sommare tutti gli elementi di un array, si può usare un ciclo for per scorrere gli elementi e sommarli, senza dover scrivere singolarmente il codice per ogni elemento.

Manipolazione dei dati: gli array offrono la possibilità di manipolare i dati in vari modi, ad esempio ordinandoli o filtrandoli.

In generale, gli array sono un'ottima scelta quando si devono gestire grandi quantità di dati di tipo omogeneo. Tuttavia, ci sono anche situazioni in cui può essere più conveniente utilizzare altre strutture dati, come le collezioni, a seconda delle esigenze specifiche del programma.

I vantaggi nell'uso delle collezioni sono:

Flessibilità: le collezioni forniscono una vasta gamma di strutture dati, come liste, insiemi e mappe, che possono essere scelte in base alle esigenze specifiche del programma.

Dimensioni dinamiche: le collezioni possono espandersi o contrarsi in base alle esigenze del programma, senza dover allocare manualmente memoria per le nuove dimensioni.

Efficienza: le collezioni forniscono implementazioni ottimizzate per molte operazioni comuni, come l'inserimento e la ricerca di elementi.

Funzionalità avanzate: le collezioni offrono molte funzionalità avanzate, come la possibilità di ordinare gli elementi, di rimuoverli in base a criteri specifici, di filtrare gli elementi, di creare sottoclassi specializzate e di fornire iteratori per scorrere gli elementi.

Sicurezza: le collezioni possono fornire una maggiore sicurezza rispetto agli array, ad esempio, prevenendo l'accesso a elementi al di fuori dell'intervallo

dell'array.

In generale, le collezioni sono una scelta migliore rispetto agli array quando si devono gestire dati di tipo eterogeneo o quando è necessario effettuare operazioni complesse sui dati, come la ricerca o la rimozione di elementi. Inoltre, le collezioni offrono un'interfaccia più intuitiva rispetto agli array, il che rende il codice più leggibile e manutenibile.

COME DICHIARARE E UTILIZZARE ARRAY

In Java, gli array sono oggetti che possono contenere un insieme di valori dello stesso tipo. Ecco come dichiarare e utilizzare un array:

1. Dichiarazione di un array:

Per dichiarare un array, è necessario specificare il tipo degli elementi dell'array e la dimensione dell'array. Ad esempio, per dichiarare un array di interi di dimensione 5, si può utilizzare la seguente sintassi:

```
int[] numeri = new int[5];
```

In questo esempio, "int[]" indica che si tratta di un array di interi, mentre "new int[5]" crea un nuovo array di interi con una dimensione di 5.

2. Inizializzazione di un array:

Dopo aver dichiarato un array, è possibile inizializzarlo fornendo i valori degli elementi. Ad esempio, per inizializzare un array di interi con i valori 1, 2, 3, 4 e 5, si può utilizzare la seguente sintassi:

```
int[] numeri = new int[] {1, 2, 3, 4, 5};
```

In questo esempio, "{1, 2, 3, 4, 5}" fornisce i valori degli elementi dell'array.

3. Accesso agli elementi di un array:

Agli elementi di un array possono si può accedere utilizzando l'indice dell'elemento. L'indice degli elementi dell'array inizia da 0 e va fino alla dimensione dell'array meno 1. Ad esempio, per accedere al secondo elemento dell'array "numeri", si può utilizzare la seguente sintassi:

```
int secondoNumero = numeri[1];
```

In questo esempio, "numeri[1]" restituisce il secondo elemento dell'array "numeri".

4. Modifica degli elementi di un array:

Gli elementi di un array possono essere modificati utilizzando l'indice dell'elemento. Ad esempio, per modificare il terzo elemento dell'array "numeri" in 10, si può utilizzare la seguente sintassi:

```
numeri[2] = 10;
```

In questo esempio, "numeri[2]" indica il terzo elemento dell'array "numeri" e lo si modifica assegnandogli il valore 10.

In generale, gli array sono utili quando si deve gestire un insieme di valori dello stesso tipo. Tuttavia, l'uso degli array può diventare complesso se si cerca di gestire dati di tipo diverso o se si deve modificare la dimensione dell'array in modo dinamico. In questi casi, potrebbe essere più conveniente utilizzare una delle collezioni fornite dal framework Java.

ARRAY MULTIDIMENSIONALI

In Java, un array multidimensionale è un array che ha più di una dimensione. Ogni dimensione dell'array è rappresentata da una coppia di parentesi quadre []. Ad esempio, un array bidimensionale ha due dimensioni e viene dichiarato come segue:

```
int[][] arrayBidimensionale = new int[3][4];
```

In questo caso, l'array ha 3 righe e 4 colonne. È possibile accedere agli elementi dell'array specificando l'indice della riga e l'indice della colonna:

```
arrayBidimensionale[0][0] = 1; // primo elemento della prima riga  
arrayBidimensionale[1][2] = 5; // terzo elemento della seconda riga
```

Per accedere agli elementi dell'array, è possibile utilizzare cicli annidati, ad esempio:

```
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 4; j++) {  
        System.out.print(arrayBidimensionale[i][j] + " ");  
    }  
    System.out.println();  
}
```

Questo codice stampa tutti gli elementi dell'array in ordine di riga. È possibile creare anche array multidimensionali con più di due dimensioni, ad esempio:

```
int[][][] arrayTridimensionale = new int[2][3][4];
```

Questo crea un array tridimensionale con 2 pagine, 3 righe e 4 colonne.
L'accesso agli elementi di questo tipo di array richiede tre indici:

```
arrayTridimensionale[0][1][2] = 10; // assegna il valore 10 all'elemento in posizione [0][1][2]
```

In generale, gli array multidimensionali sono utilizzati per rappresentare strutture dati complesse, come ad esempio le matrici, le immagini o i cubi di Rubik.

ARRAYLIST

ArrayList è una classe incluso nel pacchetto `java.util` che implementa l'interfaccia List.

Gli ArrayList sono simili agli array, ma hanno la caratteristica di essere di dimensione dinamica. Ciò significa che la dimensione dell'ArrayList può essere aumentata o diminuita in modo dinamico durante l'esecuzione del programma.

Per dichiarare un ArrayList, possiamo usare la seguente sintassi:

```
ArrayList<tipo> nomeLista = new ArrayList<tipo>();
```

dove tipo indica il tipo di dati degli elementi contenuti nella lista, e nomeLista è il nome che diamo alla lista.

Ad esempio, per dichiarare una lista di interi:

```
ArrayList<Integer> numeri = new ArrayList<Integer>();
```

Per aggiungere elementi all'ArrayList, possiamo utilizzare il metodo add(), come nell'esempio seguente:

```
ArrayList<String> nomi = new ArrayList<String>();
nomi.add("Mario");
nomi.add("Luigi");
nomi.add("Principessa Peach");
```

Per accedere agli elementi dell'ArrayList, possiamo utilizzare il metodo get(), che accetta come parametro l'indice dell'elemento desiderato, come nell'esempio seguente:

```
System.out.println(nomi.size()); // stampa 3
```

Per rimuovere un elemento dall'ArrayList, possiamo utilizzare il metodo `remove()`, come nell'esempio seguente:

```
nomi.remove(1); // rimuove l'elemento con indice 1 (cioè "Luigi")
```

Inoltre, gli ArrayList offrono diverse altre funzionalità, come la possibilità di ordinare gli elementi, cercare un elemento specifico e molto altro ancora.

LINKEDLIST

LinkedList è una classe che rappresenta una lista concatenata. A differenza di un ArrayList, che utilizza un'implementazione basata su array, una LinkedList memorizza i propri elementi come nodi collegati da puntatori. Ciò significa che l'accesso agli elementi all'interno di una LinkedList è meno efficiente di quanto non lo sia per un ArrayList, ma l'inserimento e la cancellazione di elementi sono molto più veloci, specialmente se si trovano all'inizio o alla fine della lista.

Per creare una LinkedList, si può istanziare la classe LinkedList e poi aggiungere elementi con il metodo add:

```
LinkedList<String> lista = new LinkedList<String>();
lista.add("ciao");
lista.add("mondo");
```

Per accedere agli elementi della lista, si può utilizzare il metodo get

```
String primoElemento = lista.get(0);
```

Per aggiungere o rimuovere elementi dalla lista, si possono utilizzare i metodi add, addFirst, addLast, remove, removeFirst e removeLast. Ad esempio, per aggiungere un elemento all'inizio della lista:

```
lista.addFirst("hello");
```

Per scorrere tutti gli elementi della lista, si può utilizzare un ciclo for-each:

```
for (String elemento : lista) {  
    System.out.println(elemento);  
}
```

Le LinkedList sono spesso utilizzate per implementare code o pile, dove l'inserimento e la rimozione di elementi sono operazioni molto comuni e devono essere eseguite rapidamente.

HASHMAP

Le HashMap in Java sono una delle implementazioni della struttura dati mappa. Una mappa è una collezione di coppie chiave-valore, dove ogni chiave è associata a un valore specifico. In una HashMap, le chiavi e i valori possono essere di qualsiasi tipo di oggetto.

Le HashMap sono implementate utilizzando una tabella hash, ovvero una struttura dati che mappa una chiave a un valore. Invece di memorizzare i valori in un array, la HashMap calcola un valore hash per ogni chiave e utilizza questo valore per trovare l'indice corrispondente nella tabella hash.

Le HashMap hanno molteplici vantaggi, tra cui:

1. Velocità di accesso: la ricerca di una chiave in una HashMap è molto veloce, poiché la tabella hash calcola l'indice corrispondente in modo efficiente.
2. Gestione della memoria dinamica: le HashMap possono essere ridimensionate dinamicamente durante l'esecuzione del programma.
3. Flessibilità: le HashMap consentono di utilizzare qualsiasi tipo di oggetto come chiave e valore.

Ecco un esempio di come creare e utilizzare una HashMap in Java:

```

// Crea una nuova HashMap
HashMap<String, Integer> punteggio = new HashMap<>();

// Aggiunge elementi alla HashMap
punteggio.put("Mario", 75);
punteggio.put("Luigi", 90);
punteggio.put("Peach", 80);

// Accede ai valori della HashMap
int marioPunteggio = punteggio.get("Mario");
int luigiPunteggio = punteggio.get("Luigi");
int peachPunteggio = punteggio.get("Peach");

System.out.println("Mario ha un punteggio di " + marioPunteggio);
System.out.println("Luigi ha un punteggio di " + luigiPunteggio);
System.out.println("Peach ha un punteggio di " + peachPunteggio);

```

In questo esempio, abbiamo creato una HashMap che associa una stringa (nome) a un intero (punteggio). Abbiamo quindi aggiunto tre coppie chiave-valore alla HashMap, e infine abbiamo recuperato i punteggi di ogni persona utilizzando il metodo get().

HASHSET

4. HashSet è una classe che implementa l'interfaccia Set e utilizza una tabella hash per memorizzare i suoi elementi. A differenza di ArrayList o LinkedList, gli elementi in un HashSet non sono ordinati.
5. L'implementazione di HashSet utilizza una funzione hash per calcolare l'indice di un elemento nella tabella hash. Questo rende l'accesso e la ricerca degli elementi molto efficienti, in quanto non è necessario scorrere l'intera collezione.
6. Inoltre, HashSet garantisce che non ci siano duplicati nella collezione, in quanto la tabella hash non può contenere due elementi con lo stesso valore hash.

7. HashSet è una scelta ideale quando si desidera mantenere una collezione di elementi unici e si ha bisogno di effettuare frequenti ricerche per verificare la presenza di un elemento. Tuttavia, se è necessario mantenere un ordine specifico degli elementi, è necessario utilizzare una collezione diversa, come ad esempio LinkedHashSet.
8. HashSet utilizza una funzione hash per calcolare l'indice di un elemento nella tabella hash. Questa funzione deve essere ben definita per garantire una distribuzione uniforme degli elementi nella tabella, al fine di ottenere le prestazioni migliori possibili.
9. Per calcolare la funzione hash, ogni elemento viene trasformato in un valore numerico attraverso il metodo hashCode(), definito nella classe dell'elemento. Questo valore numerico viene poi utilizzato come indice nella tabella hash.
10. Se due elementi hanno lo stesso valore hash, vengono considerati uguali dalla HashSet. Pertanto, per garantire che la HashSet funzioni correttamente, è necessario garantire che gli elementi implementino correttamente il metodo hashCode() e il metodo equals().
11. HashSet non garantisce un ordine specifico degli elementi. Gli elementi possono essere restituiti in qualsiasi ordine durante l'iterazione della HashSet.
12. HashSet non è sincronizzato, il che significa che non è thread-safe. Se più thread accedono contemporaneamente a una HashSet, potrebbe essere necessario utilizzare metodi sincronizzati o utilizzare una collezione thread-safe come ConcurrentHashMap.
13. HashSet è efficiente nell'aggiunta, rimozione e ricerca di elementi. L'aggiunta e la rimozione di un elemento richiede tempo costante O(1), mentre la ricerca di un elemento richiede tempo proporzionale alla dimensione della HashSet O(n).
14. HashSet supporta l'iterazione sugli elementi utilizzando un iteratore o il metodo forEach().

ESERCIZIO 7

Scrivere un programma Java che chiede all'utente di inserire 10 numeri interi positivi e li memorizza in un array. Il programma deve poi calcolare e stampare la somma dei numeri pari e dei numeri dispari.

Esempio di output:

```
Inserisci un numero intero positivo: 5 Inserisci un numero intero positivo: 7  
Inserisci un numero intero positivo: 8 Inserisci un numero intero positivo: 3  
Inserisci un numero intero positivo: 10 Inserisci un numero intero positivo:  
2 Inserisci un numero intero positivo: 4 Inserisci un numero intero positivo:  
1 Inserisci un numero intero positivo: 6 Inserisci un numero intero positivo:  
9
```

La somma dei numeri pari è: 30 La somma dei numeri dispari è: 25

Soluzione:

```
1 package javabase.capitolo4;
2
3 import java.util.Scanner;
4
5 public class ArraySum {
6
7     public static void main(String[] args) {
8
9         Scanner input = new Scanner(System.in);
10        int[] numeri = new int[10];
11        int sommaPari = 0;
12        int sommaDispari = 0;
13
14        for(int i = 0; i < numeri.length; i++) {
15            System.out.print("Inserisci un numero intero positivo: ");
16            numeri[i] = input.nextInt();
17            if(numeri[i] % 2 == 0) {
18                sommaPari += numeri[i];
19            } else {
20                sommaDispari += numeri[i];
21            }
22        }
23
24        System.out.println("La somma dei numeri pari è: " + sommaPari);
25        System.out.println("La somma dei numeri dispari è: " + sommaDispari);
26
27    }
28
29 }
```

In questo esempio, si utilizza un array di interi di dimensione 10 per memorizzare i numeri inseriti dall'utente. Si utilizza un ciclo for per chiedere all'utente di inserire i numeri e per calcolare la somma dei numeri pari e dei numeri dispari. Si utilizza poi il metodo println per stampare i risultati.

ESERCIZIO 8

Scrivi un programma che prende in input una lista di parole e le inserisce in una collezione di tipo HashSet. Il programma deve poi stampare a video la lista delle parole inserite, ordinata in ordine alfabetico.

Esempio di output:

```
Inserisci una parola (q per uscire): ciao
Inserisci una parola (q per uscire): come
Inserisci una parola (q per uscire): stai
Inserisci una parola (q per uscire): q

Ecco la lista delle parole inserite:
come
ciao
stai
```

Soluzione:

```
1 package javabase.capitolo4;
2
3import java.util.HashSet;
4import java.util.Scanner;
5import java.util.Set;
6import java.util.TreeSet;
7
8public class CollectionExercise {
9
10    public static void main(String[] args) {
11        Scanner input = new Scanner(System.in);
12        Set<String> parole = new HashSet<String>();
13
14        System.out.println("Inserisci una parola (q per uscire):");
15        String parola = input.nextLine();
16        while (!parola.equals("q")) {
17            parole.add(parola);
18            System.out.println("Inserisci una parola (q per uscire):");
19            parola = input.nextLine();
20        }
21
22        Set<String> paroleOrdinate = new TreeSet<String>(parole);
23        System.out.println("\nEcco la lista delle parole inserite:");
24        for (String p : paroleOrdinate) {
25            System.out.println(p);
26        }
27    }
28}
29}
```

Nel codice sopra, utilizziamo una HashSet per memorizzare le parole inserite dall'utente e un TreeSet per ordinare le parole in ordine alfabetico. Iteriamo poi sulla TreeSet e stampiamo le parole a video.

ESERCIZIO 9

Scrivere un programma Java che legga da input una sequenza di parole (terminata da una parola vuota), le memorizzi in una ArrayList e le stampi in ordine inverso.

Esempio:

```
Inserisci una parola: Ciao  
Inserisci una parola: Mondo  
Inserisci una parola: Java  
Inserisci una parola: Programmazione  
Inserisci una parola:  
  
La lista al contrario è:  
Programmazione  
Java  
Mondo  
Ciao
```

Soluzione:

```
1 package javabase.capitolo4;
2
3 import java.util.ArrayList;
4 import java.util.Scanner;
5
6 public class ReverseArrayList {
7     public static void main(String[] args) {
8         Scanner input = new Scanner(System.in);
9
10        ArrayList<String> parole = new ArrayList<>();
11
12        System.out.println("Inserisci una parola: ");
13        String parola = input.nextLine();
14
15        while (!parola.equals("")) {
16            parole.add(parola);
17            System.out.println("Inserisci una parola: ");
18            parola = input.nextLine();
19        }
20
21        System.out.println("\nLa lista al contrario è:");
22        for (int i = parole.size() - 1; i >= 0; i--) {
23            System.out.println(parole.get(i));
24        }
25    }
26 }
```

Il programma utilizza un oggetto Scanner per leggere le parole da input. Viene creata una nuova ArrayList chiamata parole e, tramite un ciclo while, le parole inserite vengono aggiunte all'array finché non viene inserita una parola vuota.

Successivamente, tramite un ciclo for si scorre l'ArrayList in ordine inverso e si stampa ciascuna parola.

GESTIONE DELLE ECCEZIONI

In programmazione, le eccezioni sono eventi anomali che si verificano durante l'esecuzione di un programma. Quando si verifica un'eccezione, il flusso normale del programma viene interrotto e viene generato un oggetto eccezione che può essere catturato e gestito.

Le eccezioni possono essere causate da diversi fattori, come ad esempio errori di sintassi, problemi di input, errori di connessione o problemi di memoria.

Le eccezioni sono una parte importante della gestione degli errori in Java e consentono di scrivere codice più robusto e affidabile, in grado di gestire eventuali problemi che si possono verificare durante l'esecuzione del programma.

In Java, le eccezioni sono un meccanismo per gestire gli errori che possono verificarsi durante l'esecuzione del programma. Invece di terminare il programma in modo anomalo, è possibile catturare e gestire le eccezioni in modo da garantire che il programma possa continuare ad eseguire le sue attività.

In Java, le eccezioni sono rappresentate da oggetti che estendono la classe "Throwable". Esistono due tipi di eccezioni: le eccezioni controllate e le eccezioni non controllate. Le eccezioni controllate sono quelle che devono essere gestite all'interno del codice, come ad esempio le eccezioni di input/output o le eccezioni di connessione al database. Le eccezioni non controllate sono quelle che possono essere gestite a livello di programmazione, ma non sono obbligatorie, come ad esempio le eccezioni di divisione per zero o le eccezioni di tipo non valido.

Quando si verifica un

'eccezione, Java genera un oggetto eccezione che viene lanciato. Per gestire un'eccezione, è possibile utilizzare un blocco "try-catch". Il blocco "try" contiene il codice che può generare un'eccezione, mentre il blocco "catch" viene utilizzato per catturare l'eccezione e gestirla.

Ad esempio, il seguente codice utilizza un blocco "try-catch" per gestire l'eccezione di divisione per zero:

```
try {
    int result = 10 / 0; // Genera una eccezione di divisione per zero
} catch (ArithmetricException e) {
    System.out.println("Errore: " + e.getMessage()); // Stampa un messaggio di errore
}
```

In questo caso, quando viene eseguita la divisione per zero, viene generata un'eccezione di tipo "ArithmetricException". Il blocco "catch" cattura l'eccezione e stampa un messaggio di errore.

Inoltre, è possibile definire eccezioni personalizzate creando classi che estendono la classe "Exception" o la classe "RuntimeException". Ciò consente di creare eccezioni personalizzate che possono essere utilizzate all'interno del codice per gestire situazioni specifiche.

In generale, le eccezioni sono un meccanismo importante per garantire la robustezza e l'affidabilità del codice Java, in quanto consentono di gestire eventuali problemi che possono verificarsi durante l'esecuzione del programma.

COME GESTIRE LE ECCEZIONI IN JAVA

In Java, le eccezioni possono essere gestite tramite il meccanismo di try-catch. In sostanza, il blocco di codice che potrebbe generare un'eccezione viene inserito all'interno di un blocco try. Se si verifica un'eccezione, il controllo viene passato al blocco catch corrispondente.

Ecco un esempio:

```
try {  
    // codice che potrebbe generare un'eccezione  
} catch (ExceptionType1 e1) {  
    // gestione dell'eccezione di tipo ExceptionType1  
} catch (ExceptionType2 e2) {  
    // gestione dell'eccezione di tipo ExceptionType2  
} finally {  
    // codice che viene eseguito sempre alla fine del blocco try-catch  
}
```

Il blocco catch gestisce l'eccezione di un determinato tipo. Ad esempio, se viene generata un'eccezione di tipo NullPointerException, il controllo viene passato al blocco catch corrispondente a NullPointerException.

È possibile avere più di un blocco catch per gestire diversi tipi di eccezioni. In questo caso, i blocchi catch vengono definiti in ordine di specificità.

Il blocco finally viene eseguito sempre alla fine del blocco try-catch, indipendentemente dal fatto che sia stata generata o meno un'eccezione. È utile per rilasciare risorse che sono state allocate all'interno del blocco try.

Un'alternativa al blocco catch è quello di dichiarare il metodo che potrebbe generare un'eccezione con la clausola throws. In questo caso, si indica che il metodo può generare un'eccezione di un determinato tipo, ma la gestione dell'eccezione viene delegata al chiamante del metodo.

Ad esempio:

```
public void metodoChePotrebbeGenerareUnEccezione() throws ExceptionType {  
    // codice che potrebbe generare un'eccezione di tipo ExceptionType  
}
```

In generale, è una buona pratica gestire le eccezioni nel modo più specifico possibile. Ciò significa che si dovrebbe cercare di gestire l'eccezione di un tipo specifico, piuttosto che catturare tutte le eccezioni con un blocco catch generico. In questo modo, si può scrivere del codice più robusto e facilmente manutenibile.

LANCIA E CATTURA

"Lanciare" un'eccezione significa segnalare al chiamante di un metodo che si è verificata una situazione anomala. Per lanciare un'eccezione, si utilizza la parola chiave "throw" seguita dall'istanza dell'eccezione che si vuole lanciare. Ad esempio:

```
public void metodoCheLanciaUnEccezione() throws EccezionePersonalizzata {  
    if (condizioneNonSoddisfatta) {  
        throw new EccezionePersonalizzata("Messaggio di errore");  
    }  
}
```

"Catturare" un'eccezione significa gestirla in modo da evitare che il programma termini in modo anomalo. Per catturare un'eccezione, si utilizza la parola chiave "try" seguita dal blocco di codice che potrebbe generare un'eccezione, e poi il blocco "catch" seguito dal tipo di eccezione che si vuole gestire e dal codice che la gestirà. Ad esempio:

```
try {  
    metodoCheLanciaUnEccezione();  
} catch (EccezionePersonalizzata e) {  
    System.out.println("Eccezione catturata: " + e.getMessage());  
}
```

In questo esempio, se il metodo "metodoCheLanciaUnEccezione" lancia un'eccezione di tipo "EccezionePersonalizzata", verrà catturata dal blocco "catch" e stampato il messaggio di errore associato all'eccezione.

FINALLY

Il blocco finally è utilizzato per contenere il codice che viene eseguito sempre alla fine di un blocco try-catch, indipendentemente dal fatto che si verifichi o meno un'eccezione.

Il blocco finally è utile per contenere codice che deve essere eseguito sempre, ad esempio per rilasciare risorse come connessioni al database o file aperti, indipendentemente dal fatto che si verifichi o meno un'eccezione.

Il blocco finally è opzionale, ma se viene utilizzato, viene eseguito dopo che il blocco try o catch viene completato. Il codice all'interno del blocco finally viene eseguito dopo il blocco try o catch, indipendentemente dal fatto che si verifichi o meno un'eccezione.

Ecco un esempio di utilizzo del blocco finally:

```
FileInputStream fis = null;
try {
    fis = new FileInputStream("myfile.txt");
    // esegui operazioni sul file
} catch (IOException e) {
    System.out.println("Errore: " + e.getMessage());
} finally {
    try {
        if (fis != null) {
            fis.close();
        }
    } catch (IOException e) {
        System.out.println("Errore nella chiusura del file:");
    }
}
```

In questo esempio, viene aperto un file myfile.txt nel blocco try. Se si verifica un'eccezione durante l'elaborazione del file, viene gestita nel blocco catch. Nel blocco finally, viene verificato se il file è stato aperto correttamente e se è aperto, viene chiuso. In caso di errore durante la chiusura del file, viene gestita un'altra eccezione nel blocco catch interno.

STACK TRACE

Lo stack trace è un report che viene generato quando si verifica un'eccezione in un programma Java. Contiene una traccia degli stack frame, ovvero la sequenza di chiamate di metodo che ha portato all'eccezione.

Lo stack trace viene utilizzato per identificare l'origine dell'eccezione e per risolvere i problemi del programma. Nel report, ogni riga corrisponde a un frame dello stack, che indica il metodo e la classe in cui si è verificata l'eccezione. La riga superiore rappresenta il frame più recente, ovvero il metodo che ha chiamato il metodo sottostante, mentre l'ultima riga rappresenta il frame più antico, ovvero il metodo che è stato chiamato per primo.

Il report può essere stampato sulla console o scritto in un file di log per la successiva analisi. In questo modo, gli sviluppatori possono individuare l'errore e risolverlo rapidamente. Il report dello stack trace contiene informazioni dettagliate sull'eccezione, inclusi il messaggio di errore, la classe dell'eccezione e la riga di codice in cui si è verificata l'eccezione.

Il formato dello stack trace può variare a seconda dell'ambiente in cui viene eseguito il codice, ma in generale ogni riga dello stack trace indica il nome del metodo, la classe che lo contiene e il numero di riga in cui si trova il metodo. Inoltre, le righe dello stack trace sono elencate dall'ultima chiamata di metodo eseguita alla prima chiamata di metodo eseguita.

Ad esempio, supponiamo di avere un'eccezione `NullPointerException` sollevata dal metodo `doSomething` nella classe `MyClass` alla riga 42. Lo stack trace potrebbe apparire come segue:

```
Exception in thread "main" java.lang.NullPointerException  
at com.example.MyClass.doSomething(MyClass.java:42)  
at com.example.Main.main(Main.java:10)
```

La prima riga indica il tipo di eccezione e la thread in cui si è verificata. Le righe successive elencano i metodi che sono stati eseguiti prima dell'eccezione. Nel nostro esempio, il metodo doSomething in MyClass è stato chiamato alla riga 42, e a sua volta il metodo main in Main.java è stato chiamato alla riga 10.

La lettura di uno stack trace può aiutare a identificare la causa dell'eccezione e la posizione nel codice in cui è stata sollevata. Inoltre, può fornire utili informazioni di debug per aiutare a risolvere il problema.

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

La programmazione orientata agli oggetti (OOP) è un paradigma di programmazione che si basa sulla creazione di oggetti, che rappresentano entità del mondo reale o concetti astratti, e sulla manipolazione di questi oggetti tramite l'invocazione di metodi. L'OOP ha come obiettivo la modellizzazione del mondo reale in modo da poter creare software che rispecchi fedelmente la realtà.

I concetti fondamentali dell'OOP sono l'incapsulamento, l'ereditarietà e il polimorfismo. L'incapsulamento si riferisce alla capacità di raggruppare i dati e i metodi che operano su di essi all'interno di un oggetto, impedendo che vengano modificati da altre parti del programma. L'ereditarietà permette di creare nuove classi basate su altre già esistenti, ereditandone le caratteristiche e le funzionalità e aggiungendone di nuove. Il polimorfismo consente di usare gli stessi metodi per oggetti di classi diverse, a patto che condividano la stessa interfaccia.

La programmazione orientata agli oggetti ha diversi vantaggi, tra cui la modularità, la riutilizzabilità del codice, la facilità di manutenzione e l'estensibilità del software. Inoltre, l'OOP permette di creare applicazioni più flessibili e adattabili, in grado di evolversi e crescere in modo organico insieme alle esigenze dell'utente.

Java è un linguaggio di programmazione orientato agli oggetti, in cui tutte le entità del programma sono oggetti o classi. Java è stato progettato per facilitare la programmazione orientata agli oggetti, fornendo una serie di funzionalità per supportare l'incapsulamento, l'ereditarietà e il polimorfismo, come ad esempio le classi, gli oggetti, le interfacce, l'ereditarietà multipla virtuale e il garbage collection.

La programmazione orientata agli oggetti (POO) è una metodologia di programmazione che si concentra sul concetto di oggetto, ovvero una struttura dati che contiene sia dati che metodi per manipolare quei dati.

Questo paradigma di programmazione è stato introdotto negli anni '60, ma è stato sviluppato e raffinato negli anni '80 e '90.

Il linguaggio di programmazione Simula, sviluppato in Norvegia nel 1962, è considerato il primo linguaggio di programmazione orientato agli oggetti. Il linguaggio C++ è stato sviluppato negli anni '80 da Bjarne Stroustrup come estensione del linguaggio C, e ha introdotto molte delle funzionalità che ora consideriamo tipiche della POO, come l'ereditarietà e il polimorfismo. Nel 1991, il linguaggio di programmazione Java è stato sviluppato da James Gosling e il suo team alla Sun Microsystems. Java è diventato uno dei linguaggi di programmazione più popolari al mondo e ha reso la programmazione orientata agli oggetti più accessibile grazie alla sua sintassi semplice e alla gestione automatica della memoria.

Oggi, la programmazione orientata agli oggetti è un paradigma di programmazione ampiamente utilizzato in molti linguaggi di programmazione, come Java, C++, Python e Ruby, e viene utilizzato per sviluppare una vasta gamma di applicazioni, tra cui applicazioni desktop, applicazioni web e app per dispositivi mobili. La POO offre numerosi vantaggi, come la modularità, l'astrazione e la riutilizzabilità del codice, che la rendono una delle metodologie di programmazione più popolari e utilizzate.

La differenza principale tra la programmazione orientata agli oggetti (POO) e la programmazione procedurale è la gestione dei dati e delle funzioni. In un programma procedurale, i dati e le funzioni sono separati e le funzioni agiscono sui dati passati come argomenti. In POO, invece, i dati e le funzioni sono raggruppati in classi e gli oggetti vengono creati a partire da queste classi.

In POO, gli oggetti rappresentano entità del mondo reale o astratto e contengono sia dati che le funzioni che agiscono su quei dati. In questo modo, la POO consente di creare programmi più modulari, scalabili e facili da mantenere. La POO inoltre supporta la programmazione modulare e la riusabilità del codice, in quanto gli oggetti possono essere creati a partire dalle classi esistenti.

La programmazione procedurale, invece, si concentra sull'elaborazione di funzioni e procedure che manipolano i dati, spesso organizzati in variabili globali o locali. In questa modalità, il codice è spesso organizzato in

funzioni che agiscono sui dati, e queste funzioni possono essere richiamate in qualsiasi momento nel programma.

In sintesi, la principale differenza tra la POO e la programmazione procedurale sta nell'organizzazione dei dati e delle funzioni, che nella POO sono raggruppati in classi e oggetti, mentre nella programmazione procedurale sono organizzati in funzioni e variabili.

La programmazione ad oggetti è stata sviluppata con l'idea di modellare il mondo reale e di fornire un paradigma di programmazione che fosse più intuitivo e vicino al modo in cui le persone pensano e ragionano.

A differenza della programmazione procedurale, che si basa principalmente su una sequenza di istruzioni per risolvere un problema, la programmazione ad oggetti si concentra sulle entità che compongono un sistema e sulle relazioni tra di esse. Queste entità sono rappresentate come oggetti, che possono avere proprietà (variabili) e metodi (funzioni) che possono essere chiamati per eseguire azioni o restituire informazioni.

Ad esempio, se si vuole rappresentare un'automobile in un sistema di programmazione ad oggetti, si potrebbe creare una classe Automobile che ha proprietà come la marca, il modello, il colore, la velocità attuale e i metodi come accelerare, frenare e girare a sinistra/destra. Questo approccio consente di modellare il mondo reale in modo più naturale e di creare sistemi più complessi e flessibili.

Inoltre, la programmazione ad oggetti favorisce la modularità e la riusabilità del codice, in quanto gli oggetti possono essere creati una sola volta e utilizzati in più parti del programma. In questo modo, si evita la ripetizione di codice e si riducono gli errori di programmazione.

In generale, la programmazione ad oggetti è considerata più espressiva e potente rispetto alla programmazione procedurale e ha portato a importanti sviluppi nel campo dell'informatica, come la creazione di framework e librerie di codice riutilizzabile.

CONCETTI DI CLASSE E OGGETTO

In Java, una classe è una struttura che descrive le proprietà e i comportamenti di un insieme di oggetti. In altre parole, una classe definisce un tipo di dati personalizzato. Gli oggetti sono le istanze di una classe, cioè le entità che vengono create e manipolate durante l'esecuzione del programma.

Una classe in Java contiene due tipi di membri: campi (o attributi) e metodi. I campi sono variabili che descrivono lo stato dell'oggetto, mentre i metodi sono funzioni che definiscono il comportamento dell'oggetto.

Ad esempio, possiamo definire una classe "Persona" con i seguenti campi: nome, cognome, data di nascita e indirizzo, e con i seguenti metodi: `getNome`, `getCognome`, `getDataNascita`, `getIndirizzo`, `setNome`, `setCognome`, `setDataNascita` e `setIndirizzo`. In questo modo, possiamo creare istanze della classe "Persona" per rappresentare persone specifiche nel nostro programma.

Per creare un oggetto in Java, dobbiamo prima definire la classe corrispondente. Una volta definita la classe, possiamo creare un'istanza dell'oggetto utilizzando la parola chiave "new". Ad esempio, possiamo creare un'istanza della classe "Persona" con il seguente codice:

```
Persona p = new Persona();
p.setNome("Mario");
p.setCognome("Rossi");
p.setDataNascita("01/01/1990");
p.setIndirizzo("Via Roma, 1");
```

In questo esempio, abbiamo creato un'istanza della classe "Persona" utilizzando la parola chiave "new", e poi abbiamo impostato i campi dell'oggetto utilizzando i metodi setter definiti nella classe.

Una classe è come uno stampo o un modello che descrive come un oggetto dovrebbe essere formato e come dovrebbe comportarsi, mentre un oggetto è l'istanza concreta di quella classe, con i suoi dati specifici e i comportamenti che gli sono stati assegnati.

INCAPSULAMENTO

L'incapsulamento è uno dei concetti fondamentali della programmazione orientata agli oggetti e si riferisce alla capacità di una classe di nascondere i dettagli interni dell'implementazione e di fornire solo un'interfaccia pubblica per l'interazione con gli oggetti di quella classe. In altre parole, l'incapsulamento permette di proteggere i dati e le operazioni all'interno di una classe, in modo che possano essere utilizzati solo attraverso metodi pubblici specifici.

L'incapsulamento aiuta a migliorare la sicurezza e la stabilità del codice, poiché previene l'accesso diretto ai dati e alle operazioni di una classe da parte di altre parti del programma. Inoltre, l'incapsulamento facilita la modifica dell'implementazione interna di una classe senza dover modificare l'interfaccia pubblica, il che consente di mantenere la compatibilità con il codice già esistente che utilizza la classe.

In Java, l'incapsulamento viene implementato attraverso l'utilizzo di modificatori di accesso, come **public**, **private** e **protected**, che determinano quali parti del codice possono accedere ai campi e ai metodi di una classe. I campi di una classe dovrebbero essere dichiarati privati, mentre i metodi che operano su quei campi dovrebbero essere dichiarati pubblici o protetti. In questo modo, i dati sono accessibili solo attraverso i metodi pubblici e la classe stessa può controllare l'accesso ai dati per garantire la loro integrità e coerenza.

In Java, i modificatori sono parole chiave utilizzate per specificare l'accesso, la visibilità e altre proprietà delle classi, dei metodi, dei campi e dei costruttori. I modificatori possono essere suddivisi in quattro categorie principali:

Modificatori di accesso: specificano l'accessibilità di una classe, di un metodo o di un campo ai vari livelli di visibilità (public, private, protected e default).

public: la classe, il metodo o il campo è accessibile da qualsiasi parte del codice.

private: la classe, il metodo o il campo è accessibile solo all'interno della classe in cui è dichiarato.

protected: la classe, il metodo o il campo è accessibile all'interno della classe e dalle classi derivate nella stessa gerarchia di ereditarietà.

default: la classe, il metodo o il campo è accessibile all'interno del pacchetto in cui è dichiarato.

Modificatori di non accesso: specificano altre proprietà di una classe, di un metodo o di un campo.

final: la classe, il metodo o il campo non può essere esteso o sovrascritto.

abstract: la classe o il metodo non può essere istanziato o deve essere sovrascritto dalle sottoclassi.

static: il campo o il metodo appartiene alla classe anziché alle istanze e può essere richiamato direttamente dalla classe stessa.

synchronized: il metodo può essere eseguito da un solo thread alla volta.

volatile: il valore del campo può essere modificato da più thread contemporaneamente.

Modificatori di classificazione: specificano la classificazione di una classe come classe **enum**, classe annotata, classe interfaccia o classe record.

enum: dichiara una classe enum che rappresenta un tipo di dato con un insieme predefinito di costanti.

annotation: dichiara una classe annotazione che rappresenta un'annotazione da utilizzare su classi, metodi o campi.

interface: dichiara una classe interfaccia che rappresenta un contratto che una classe deve implementare.

record: dichiara una classe record, una nuova funzionalità introdotta in Java 16, che consente di dichiarare rapidamente classi di dati immutabili.

Modificatori di altri contesti: specificano modifiche a classi, metodi o campi in contesti specifici.

transient: il campo non viene serializzato quando l'oggetto viene salvato o trasferito.

native: il metodo è implementato in codice nativo di una piattaforma specifica.

strictfp: il metodo segue lo standard IEEE 754 per la precisione dei calcoli in virgola mobile.

Questi sono solo alcuni esempi di modificatori disponibili in Java, ma ci sono molti altri che possono essere utilizzati per personalizzare la visibilità e il comportamento di una classe, di un metodo o di un campo.

EREDITARIETÀ

In Java, l'ereditarietà è un meccanismo che permette ad una classe di ereditare tutti i campi e i metodi pubblici e protetti di un'altra classe, chiamata superclasse o classe genitore. La classe che eredita i campi e i metodi è chiamata sottoclasse o classe figlio.

Per creare una sottoclasse in Java, si utilizza la parola chiave "extends" seguita dal nome della superclasse. La sottoclasse eredita tutti i campi e i metodi pubblici e protetti della superclasse, ma non quelli privati. La sottoclasse può quindi utilizzare questi campi e metodi come se fossero stati definiti nella stessa classe.

Ecco un esempio di come si può definire una sottoclasse in Java:

```
public class Persona {  
    private String nome;  
    private int eta;  
    public Persona(String nome, int eta) {  
        this.nome = nome;  
        this.eta = eta;  
    }  
    public String getNome() {  
        return nome;  
    }  
    public int getEta() {  
        return eta;  
    }  
}  
  
public class Studente extends Persona {  
    private int matricola;  
    public Studente(String nome, int eta, int matricola) {  
        super(nome, eta);  
        this.matricola = matricola;  
    }  
    public int getMatricola() {  
        return matricola;  
    }  
}
```

In questo esempio, la classe Studente estende la classe Persona. La classe Persona ha due campi privati nome ed eta, e un costruttore che li inizializza. La classe Studente ha un campo privato matricola, un costruttore che richiama il costruttore del superclasse con la parola chiave super, e un metodo getMatricola che restituisce la matricola dello studente.

L'ereditarietà può essere utile per evitare la duplicazione di codice tra classi simili, o per specializzare una classe esistente. Tuttavia, bisogna fare attenzione a non utilizzare troppa ereditarietà, in quanto può portare ad una gerarchia di classi complicata e difficile da gestire. Inoltre, l'ereditarietà può rendere il codice meno flessibile, in quanto le modifiche al superclasse possono avere effetti sulla sottoclasse.

L'ereditarietà singola (o single inheritance) è un principio della programmazione orientata agli oggetti in cui una classe figlia (o sottoclasse) può ereditare i membri della classe genitore (o superclasse) e implementare nuovi membri specifici per la classe figlia. In Java, ogni classe può avere al massimo un superclasse diretta, il che significa che l'ereditarietà è singola.

Ciò significa che ogni classe in Java, ad eccezione della classe di base Object, ha una sola superclasse diretta. La classe figlia può ereditare tutti i membri non privati della classe genitore, tra cui campi, metodi e altre proprietà. Inoltre, la classe figlia può aggiungere nuovi membri, ridefinire i membri ereditati e fornire implementazioni uniche per i metodi ereditati.

Ad esempio, supponiamo di avere una classe chiamata "Veicolo" che ha proprietà come "velocità massima" e "numero di posti", e una classe figlia chiamata "Automobile" che eredita le proprietà di Veicolo e aggiunge altre proprietà specifiche come "marca" e "modello". In questo caso, Automobile è una sottoclasse di Veicolo e Veicolo è il superclasse di Automobile.

Java utilizza l'ereditarietà singola per motivi di chiarezza e semplicità del codice. L'ereditarietà singola consente di creare gerarchie di classi chiare e facili da comprendere, in cui ogni classe ha un unico genitore (classe padre) e una serie di figli (classi figlie). In questo modo, si evita la complessità di gerarchie multiple, in cui una classe potrebbe ereditare da più di una classe padre, creando confusioni e ambiguità nella gestione dei metodi e degli attributi ereditati. Inoltre, l'ereditarietà singola consente di mantenere la coerenza e la modularità del codice, semplificando la sua manutenzione e la sua evoluzione nel tempo.

Vi sono vari modi per scrivere le classi all'interno dei file java, come abbiamo visto il più comune e più chiaro è quello di dichiarare una classe all'interno di un file che avrà lo stesso nome, è però possibile scrivere più classi all'interni di un singolo file java con l'utilizzo delle, le inner class (o classi interne) sono classi definite all'interno di un'altra classe. Ci sono quattro tipi di inner class:

In Java, le inner class (o classi interne) sono classi definite all'interno di un'altra classe. Ci sono quattro tipi di inner class:

1. Member Inner Class: è una classe definita all'interno di un'altra classe e può accedere a tutti i membri della classe esterna, inclusi quelli privati.
2. Local Inner Class: è una classe definita all'interno di un metodo o di un blocco di codice e può accedere ai membri della classe esterna solo se questi sono final o effettivamente final.
3. Anonymous Inner Class: è una classe senza nome definita all'interno di un'altra classe o di un metodo e viene istanziata immediatamente.
4. Static Nested Class: è una classe statica definita all'interno di un'altra classe e non ha accesso ai membri non statici della classe esterna.

Le inner class sono utilizzate per raggruppare logicamente le classi e migliorare l'incapsulamento, consentendo alle classi interne di accedere ai membri privati della classe esterna. Inoltre, possono essere utilizzate per implementare interfacce o classi astratte all'interno di una classe esterna e per creare classi annidate per implementare strutture dati complesse.

```

public class Main {
    public static void main(String[] args) {
        // Istanza di InnerClass all'interno di OuterClass
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        System.out.println(inner.myInnerMethod()); // output: 10

        // Istanza di InnerClass al di fuori di OuterClass
        OuterClass.InnerClass inner2 = new OuterClass().new InnerClass();
        System.out.println(inner2.myInnerMethod()); // output: 10
    }
}

```

Nell'esempio, creiamo un'istanza della classe interna all'interno della classe esterna e accediamo al metodo myInnerMethod(), che restituisce il valore di x. Possiamo anche creare un'istanza della classe interna al di fuori della classe esterna, come mostrato nel secondo blocco di codice.

una classe anonima è una classe senza un nome esplicito, che viene definita e istanziata allo stesso tempo. Le classi anonime sono utilizzate per creare una sola istanza di una classe in-linea, senza la necessità di definire una classe separata per l'uso una sola volta.

Un esempio di classe anonima è la creazione di un oggetto ActionListener per gestire un evento di un pulsante:

```

JButton button = new JButton("Click me!");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});

```

In questo esempio, viene creato un oggetto JButton e viene definito un ActionListener anonimo che viene passato come parametro al metodo addActionListener(). L'ActionListener anonimo viene definito all'interno di

una coppia di parentesi graffe, che rappresenta la definizione della classe anonima. La classe anonima implementa l'interfaccia ActionListener e implementa il metodo actionPerformed() per gestire l'evento del pulsante. Le classi anonime sono utili quando è necessario implementare un'interfaccia o una classe astratta per un solo uso, senza la necessità di definire una classe separata. Tuttavia, le classi anonime possono rendere il codice meno leggibile, quindi è importante utilizzarle con parsimonia e solo quando necessario

POLIMORFISMO

Il polimorfismo è uno dei concetti chiave della programmazione orientata agli oggetti. Si riferisce alla capacità di oggetti di classi diverse di essere trattati come se fossero della stessa classe, ovvero la classe padre.

Il polimorfismo è spesso utilizzato insieme all'ereditarietà, dove una classe figlia può estendere una classe padre e utilizzare i metodi e le proprietà della classe padre. Questo significa che un'istanza della classe figlia può essere trattata come un'istanza della classe padre. Ad esempio, se hai una classe animale e una classe cane che estende la classe animale, un'istanza di cane può essere trattata come un'istanza di animale. Questo è possibile perché il cane eredita tutte le proprietà e i metodi della classe animale.

In Java, il polimorfismo può essere realizzato tramite l'uso di classi e metodi virtuali. Quando si definisce un metodo nella classe padre e si esegue l'override di quel metodo nella classe figlia, è possibile chiamare il metodo nella classe figlia e utilizzare la stessa chiamata del metodo nella classe padre. Ciò significa che se si utilizza un'istanza della classe figlia, il metodo della classe figlia sarà chiamato. Se si utilizza un'istanza della classe padre, il metodo della classe padre verrà chiamato.

In sintesi, il polimorfismo ci consente di scrivere codice che può funzionare con una varietà di tipi di oggetti, rendendo il nostro codice più flessibile e riutilizzabile,

Il polimorfismo si riferisce alla capacità di un oggetto di essere trattato come un altro tipo di oggetto. Ciò significa che un oggetto può assumere diversi

comportamenti a seconda del contesto in cui viene utilizzato. Il polimorfismo viene spesso implementato attraverso l'uso di metodi polimorfici o di metodi virtuali.

L'ereditarietà multipla, d'altra parte, si riferisce alla capacità di una classe di ereditare da più di una classe padre. In altre parole, una classe figlia può ereditare attributi e metodi da più di una classe padre. Mentre questa capacità può essere utile in alcune situazioni, può anche portare a problemi di ambiguità e complessità del codice.

Java non supporta l'ereditarietà multipla, il che significa che una classe può ereditare da una sola classe padre. Tuttavia, Java supporta il polimorfismo attraverso l'uso di interfacce e classi astratte, che consentono a un oggetto di essere trattato come un'istanza di un'interfaccia o di una classe astratta, rispettivamente. Ciò consente di implementare un'ampia gamma di comportamenti polimorfici senza dover utilizzare l'ereditarietà multipla, rendendo il nostro codice più flessibile e riutilizzabile.

Supponiamo di avere una classe Veicolo e due sottoclassi Automobile e Moto, che estendono la classe Veicolo. Entrambe le sottoclassi hanno un metodo muovi() che consente loro di muoversi, ma con un comportamento diverso. Ad esempio, l'Automobile può avere una velocità massima più elevata della Moto.

Utilizzando il polimorfismo, possiamo creare un array di oggetti Veicolo e inserirvi al suo interno sia oggetti Automobile che Moto, senza specificare di quale tipo specifico siano gli oggetti. Possiamo quindi chiamare il metodo muovi() su ciascun oggetto dell'array, senza dover sapere a priori di che tipo sia l'oggetto e ottenere il comportamento specifico della classe di appartenenza.

Ad esempio:

```
Veicolo[] veicoli = new Veicolo[2];
veicoli[0] = new Automobile();
veicoli[1] = new Moto();

for (Veicolo veicolo : veicoli) {
    veicolo.muovi();
}
```

In questo caso, il metodo muovi() verrà chiamato sulla classe corretta di ciascun oggetto, ovvero Automobile.muovi() per il primo oggetto e Moto.muovi() per il secondo oggetto, senza dover specificare il tipo specifico degli oggetti nel codice. Questo è un esempio di polimorfismo.

L'override in Java si riferisce alla capacità di una classe figlia di fornire una propria implementazione per un metodo già definito nella classe genitore, in modo che il metodo figlio possa essere chiamato al posto di quello genitore. In altre parole, l'override consente di ridefinire il comportamento di un metodo già presente nella classe genitore, ma in modo specifico per la classe figlio.

Per eseguire l'override di un metodo, è necessario che il nome del metodo, i tipi di parametro e il valore di ritorno siano esattamente gli stessi nella classe figlio e nella classe genitore. Inoltre, il modificatore di accesso del metodo figlio deve essere uguale o meno restrittivo del modificatore di accesso del metodo genitore.

Ecco un esempio di override in Java:

```
public class Animale {
    public void emettiVerso() {
        System.out.println("Questo animale emette un verso generico.");
    }
}

public class Cane extends Animale {
    @Override
    public void emettiVerso() {
        System.out.println("Questo cane abbaia.");
    }
}
```

Nell'esempio sopra, la classe Cane estende la classe Animale e ridefinisce il metodo emettiVerso(). Quando viene chiamato il metodo emettiVerso() su un oggetto di tipo Cane, verrà eseguita l'implementazione specifica del cane, anziché quella generica dell'animale.

INTERFACCIA

Le interfacce in Java sono un meccanismo che permette di definire un insieme di metodi astratti, che le classi che le implementano devono obbligatoriamente implementare. Le interfacce sono simili alle classi astratte, ma con alcune differenze importanti.

Una classe astratta può avere sia metodi astratti che concreti, e può avere anche attributi. Inoltre, una classe può estendere una sola classe astratta. D'altra parte, un'interfaccia può contenere solo metodi astratti e costanti, e può essere implementata da molte classi diverse. Una classe può implementare più di un'interfaccia.

Le interfacce possono essere utilizzate per definire una specifica di come un oggetto dovrebbe comportarsi, senza dover specificare l'implementazione effettiva. Ad esempio, si potrebbe definire un'interfaccia "Veicolo" che ha i metodi "accelerare", "frenare" e "girare". Una classe "Automobile" e una classe "Moto" potrebbero entrambe implementare l'interfaccia "Veicolo", ma con implementazioni diverse dei metodi.

Le interfacce in Java possono anche contenere metodi predefiniti (default methods) e metodi statici. I metodi predefiniti sono metodi che hanno un'implementazione predefinita all'interno dell'interfaccia stessa, mentre i metodi statici sono metodi che possono essere chiamati senza creare un'istanza dell'oggetto che implementa l'interfaccia.

Le interfacce in Java sono spesso utilizzate per definire API (Application Programming Interface) per le librerie o i framework. Ad esempio, le interfacce "List", "Set" e "Map" sono definite nella libreria standard di Java e vengono implementate da molte classi diverse, come ArrayList, HashSet e HashMap, per fornire un'implementazione specifica dei metodi definiti nelle interfacce.

In sintesi, le interfacce in Java sono un meccanismo importante per definire un contratto tra le classi e le librerie, e per garantire la compatibilità tra le diverse implementazioni.

In Java, si può definire un'interfaccia utilizzando la parola chiave "interface". Ecco un esempio di come definire un'interfaccia:

```
public interface MyInterface {  
    public void myMethod();  
}
```

In questo esempio, abbiamo definito un'interfaccia chiamata "MyInterface" che ha un metodo chiamato "myMethod" senza corpo.

È importante notare che tutti i metodi definiti in un'interfaccia sono implicitamente pubblici e astratti. Gli attributi (cioè le variabili di istanza) non possono essere definiti all'interno di un'interfaccia, solo le costanti.

Inoltre, le interfacce possono anche estendere altre interfacce. Ecco un esempio:

```
public interface MyInterface extends ParentInterface {  
    // metodi dell'interfaccia  
}
```

In questo esempio, abbiamo definito un'interfaccia "MyInterface" che estende un'altra interfaccia chiamata "ParentInterface".

una classe astratta è una classe che non può essere istanziata direttamente, ma può essere utilizzata solo come modello per altre classi. Una classe astratta è una classe parzialmente implementata, in cui alcuni metodi sono dichiarati ma non implementati. Gli sviluppatori che utilizzano la classe astratta possono implementare questi metodi in sottoclassi che estendono la classe astratta.

La classe astratta fornisce una sorta di "modello" per altre classi, definendo il comportamento comune a tutte le classi che estendono quella classe. Ad esempio, si potrebbe avere una classe astratta "Veicolo" che definisce le proprietà e i comportamenti comuni a tutti i veicoli, come la velocità massima e la capacità del serbatoio del carburante. Le classi che estendono la classe astratta "Veicolo" possono quindi aggiungere proprietà e metodi specifici per i diversi tipi di veicoli, come le ruote e la potenza del motore.

Per dichiarare una classe come astratta in Java, si utilizza la parola chiave "abstract" davanti alla parola chiave "class". Ad esempio:

```
public abstract class Veicolo {  
    // proprietà e metodi comuni a tutti i veicoli  
    public abstract void accelera();  
    // dichiarazione del metodo accelera(), che deve essere implementato dalle sottoclassi  
}
```

In questo esempio, la classe "Veicolo" è dichiarata come astratta utilizzando la parola chiave "abstract". Inoltre, il metodo "accelera()" è dichiarato come astratto utilizzando la parola chiave "abstract", il che significa che deve essere implementato dalle sottoclassi che estendono la classe "Veicolo".

Le classi astratte possono anche avere metodi non astratti, che possono essere implementati direttamente nella classe astratta o lasciati vuoti e implementati solo nelle sottoclassi. Inoltre, le classi astratte possono estendere altre classi astratte, creando una gerarchia di classi astratte che possono essere estese ulteriormente dalle classi concrete.

In Java, si possono utilizzare sia le interfacce che le classi astratte come meccanismo per definire una gerarchia di classi. In generale, si utilizzano le interfacce quando si vuole specificare un comportamento comune a diverse classi, mentre si utilizzano le classi astratte quando si vuole fornire un'implementazione parziale di una classe.

Ecco alcune considerazioni generali sull'uso delle interfacce e delle classi astratte:

Interfacce: le interfacce vengono utilizzate per definire un insieme di metodi che una classe deve implementare. Si utilizzano quando si vuole garantire che diverse classi abbiano un comportamento comune, ma non si vuole specificare come questo comportamento debba essere implementato. Le interfacce possono essere utilizzate anche per definire costanti comuni a diverse classi.

Classi astratte: le classi astratte vengono utilizzate per definire una classe base che implementa alcune funzionalità comuni a diverse sottoclassi. In una classe astratta si possono definire metodi astratti, che le sottoclassi devono implementare, e metodi con implementazione concreta, che le

sottoclassi possono eventualmente sovrascrivere. Si utilizzano quando si vuole definire una gerarchia di classi, ma non si vuole implementare completamente la classe base.

In generale, si può utilizzare una classe astratta quando si vuole definire una gerarchia di classi che ha alcune funzionalità comuni, ma che richiede anche alcune differenze tra le sottoclassi. Le interfacce, d'altra parte, si utilizzano quando si vuole garantire che diverse classi abbiano un comportamento comune, ma che potrebbero avere alcune differenze nelle implementazioni.

ESERCIZIO 10

Supponiamo di voler creare un sistema di gestione di una biblioteca. Possiamo modellare il sistema usando la programmazione orientata agli oggetti in Java.

Di seguito sono riportate le specifiche per la creazione del sistema:

1. Creare una classe Libro che contenga i seguenti campi:
 - titolo (String)
 - autore (String)
 - anno (int)
 - genere (String)

La classe Libro deve avere un costruttore che inizializza i campi della classe e i relativi metodi getter e setter per accedere ai campi.

2. Creare una classe Utente che contenga i seguenti campi:
 - nome (String)
 - cognome (String)
 - id (int)
 - prestiti (ArrayList<Libro>)

La classe Utente deve avere un costruttore che inizializza i campi della classe e i relativi metodi getter e setter per accedere ai campi.

3. Creare una classe Biblioteca che contenga i seguenti campi:

- libri (ArrayList<Libro>)
- utenti (ArrayList<Utente>)

La classe Biblioteca deve avere i seguenti metodi:

- aggiungiLibro(Libro libro): aggiunge un libro all'elenco dei libri della biblioteca.
- rimuoviLibro(Libro libro): rimuove un libro dall'elenco dei libri della biblioteca.
- cercaLibro(String titolo): cerca un libro per titolo e restituisce l'elenco dei libri corrispondenti.
- aggiungiUtente(Utente utente): aggiunge un utente all'elenco degli utenti della biblioteca.
- rimuoviUtente(Utente utente): rimuove un utente dall'elenco degli utenti della biblioteca.
- cercaUtente(int id): cerca un utente per id e restituisce l'utente corrispondente.
- prestitoLibro(Libro libro, Utente utente): presta un libro a un utente e aggiunge il libro alla lista dei prestiti dell'utente.
- restituisciLibro(Libro libro, Utente utente): restituisce un libro alla biblioteca e rimuove il libro dalla lista dei prestiti dell'utente.

Ecco una possibile implementazione

```
5  public class Biblioteca {
6      private ArrayList<Libro> libri = new ArrayList<Libro>();
7      private ArrayList<Utente> utenti = new ArrayList<Utente>();
8
9      public void aggiungiLibro(Libro libro) {
10         libri.add(libro);
11     }
12
13     public void rimuoviLibro(Libro libro) {
14         libri.remove(libro);
15     }
16
17     public Libro cercaLibro(String titolo) {
18         Libro result = libri.stream().filter(libro -> titolo.equals(libro.getTitolo())).findAny().orElse(null);
19         return result;
20     }
21
22     public void aggiungiUtente(Utente utente) {
23         utenti.add(utente);
24     }
25
26     public void rimuoviUtente(Utente utente) {
27         utenti.remove(utente);
28     }
29
30     public Utente cercaUtente(int id) {
31         Utente result = utenti.stream().filter(utente -> id==utente.getId() ).findAny().orElse(null);
32         return result;
33     }
34
35     public void prestitoLibro(Libro libro,Utente utente) {
36         utente.getPrestiti().add(libro);
37         libri.remove(libro);
38     }
39
40     public void restituisceLibro(Libro libro,Utente utente) {
41         utente.getPrestiti().remove(libro);
42         libri.add(libro);
43     }
44 }
45 }
```

ESERCIZIO 11

Si vuole creare una piccola applicazione per la gestione delle prenotazioni di un albergo. Si richiede di creare le seguenti classi:

1. Prenotazione: classe astratta che rappresenta una prenotazione generica e che contiene almeno i seguenti attributi:
 - id: identificativo della prenotazione
 - data: data della prenotazione
 - numeroNotti: numero di notti della prenotazione
 - costo: costo totale della prenotazione

Inoltre, deve contenere i seguenti metodi astratti:

- calcolaCosto(): metodo che calcola il costo totale della prenotazione
 - toString(): metodo che restituisce una stringa rappresentante la prenotazione in formato testuale.
2. PrenotazioneSingola: classe che rappresenta una prenotazione per una singola camera. Questa classe estende la classe Prenotazione e contiene i seguenti attributi aggiuntivi:
 - numeroCamera: numero della camera prenotata
 - prezzoNotte: prezzo della camera a notte

Implementare i metodi astratti ereditati dalla classe Prenotazione.

3. PrenotazioneGruppo: classe che rappresenta una prenotazione per più camere. Questa classe estende la classe Prenotazione e contiene i seguenti attributi aggiuntivi:
 - numeroCamere: numero di camere prenotate
 - prezzoCamera: prezzo della camera a notte

Implementare i metodi astratti ereditati dalla classe Prenotazione.

4. Albergo: classe che rappresenta l'albergo e che contiene una lista di prenotazioni. Deve contenere i seguenti metodi:
 - aggiungiPrenotazione(Prenotazione p): metodo che aggiunge una prenotazione alla lista delle prenotazioni
 - calcolaGuadagnoTotale(): metodo che calcola il guadagno totale dell'albergo

ESERCIZIO 12

Supponiamo di avere una classe astratta Animal con un metodo makeSound() e due sottoclassi concrete: Dog e Cat. La classe Dog ridefinisce il metodo makeSound() per stampare "Woof!", mentre la classe Cat ridefinisce il metodo makeSound() per stampare "Meow!".

Vogliamo ora creare un array di animali, che possa contenere sia oggetti Dog che oggetti Cat, e per ciascun animale chiamare il metodo makeSound().

Soluzione:

```

abstract class Animal {
    abstract void makeSound();
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow!");
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Animal[] animals = new Animal[2];
        animals[0] = new Dog();
        animals[1] = new Cat();

        for (Animal animal : animals) {
            animal.makeSound();
        }
    }
}

```

In questo esempio, abbiamo dichiarato un array `animals` di tipo `Animal`, che può contenere oggetti di qualsiasi classe che estende la classe astratta `Animal`. Abbiamo creato poi due oggetti `Dog` e `Cat` e li abbiamo assegnati all'array `animals`.

Infine, abbiamo iterato attraverso l'array `animals` e chiamato il metodo `makeSound()` per ciascun animale. Grazie al polimorfismo, il metodo `makeSound()` della classe corretta (cioè `Dog` o `Cat`) viene chiamato per ciascun oggetto.

INTRODUZIONE AI DESIGN PATTERN

Definizione di Design Pattern

Un Design Pattern, in informatica, è una soluzione riusabile a un problema comune che si verifica durante la progettazione del software. Si tratta di un concetto chiave della progettazione del software orientata agli oggetti e rappresenta un modo per descrivere soluzioni comuni e collaudate a problemi ricorrenti.

Un Design Pattern fornisce un modello generale per risolvere un problema, e può essere implementato in diversi linguaggi di programmazione. Un Design Pattern fornisce inoltre un linguaggio comune per la discussione e la documentazione delle soluzioni, che consente ai progettisti di comunicare in modo più efficace.

I Design Pattern vengono utilizzati per risolvere problemi comuni come la gestione delle dipendenze tra oggetti, la gestione delle transazioni e il caching dei dati. Essi aiutano i progettisti a creare software più flessibili, scalabili e mantenibili, riducendo il costo e il rischio associati allo sviluppo di nuove applicazioni.

Esistono numerosi Design Pattern, ognuno dei quali risolve un problema specifico. Essi possono essere classificati in categorie come creazionali, strutturali e comportamentali, a seconda del problema che risolvono.

I vantaggi dei Design Pattern sono numerosi e includono:

1. Riusabilità del codice: i Design Pattern forniscono una soluzione generale e riusabile a problemi comuni di progettazione, consentendo ai programmatore di non dover reinventare la ruota ogni volta.
2. Migliore organizzazione del codice: l'uso dei Design Pattern aiuta a organizzare il codice in modo più strutturato e

comprendibile, semplificando la manutenzione e la modifica del codice.

3. Facilità di comprensione del codice: i Design Pattern hanno nomi e scopi ben definiti, rendendo più facile per gli sviluppatori comprendere ciò che il codice fa e come funziona.
4. Migliore scalabilità del sistema: i Design Pattern aiutano a creare sistemi che possono essere facilmente estesi e scalati in modo da soddisfare le esigenze future.
5. Maggiore qualità del software: l'uso dei Design Pattern porta a un codice più pulito e strutturato, che rende meno probabile la presenza di bug o errori.
6. Migliore collaborazione: l'uso di Design Pattern facilita la collaborazione tra gli sviluppatori, in quanto fornisce una terminologia comune e una soluzione condivisa ai problemi di progettazione.

ABSTRACT FACTORY

L'Abstract Factory è un Design Pattern creazionale che fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. L'idea principale di questo pattern è quella di creare un'astrazione del processo di creazione dell'oggetto in modo che le classi che lo utilizzano non dipendano dalle classi concrete degli oggetti che devono essere creati.

L'Abstract Factory si basa sull'utilizzo di un'interfaccia o di una classe astratta per definire un insieme di metodi di fabbricazione, ciascuno dei quali produce un oggetto di una specifica famiglia. Le classi concrete che implementano questa interfaccia o classe astratta sono responsabili della creazione degli oggetti specifici.

Il vantaggio principale dell'Abstract Factory è quello di creare un'interfaccia standardizzata per la creazione di oggetti, facilitando così l'interoperabilità tra le diverse parti di un sistema. Inoltre, questo pattern consente di aggiungere nuove famiglie di oggetti senza dover modificare il codice esistente, in quanto le classi che utilizzano l'Abstract Factory non sono consapevoli delle classi concrete degli oggetti che vengono creati.

In sintesi, l'Abstract Factory è un pattern utile per creare un'interfaccia comune per la creazione di famiglie di oggetti correlati, fornendo un alto livello di astrazione e modularità al processo di creazione degli oggetti.

BUILDER

Il pattern Builder è un design pattern creazionale che consente di creare oggetti complessi passo dopo passo, separando il processo di costruzione dall'oggetto finale. In altre parole, il pattern Builder è utile quando abbiamo bisogno di creare un oggetto complesso, ma non vogliamo costruirlo in una sola volta, bensì vogliamo costruirlo pezzo per pezzo.

Il pattern Builder si compone di una classe Builder, che ha il compito di costruire l'oggetto, e di una classe Product, che rappresenta l'oggetto da costruire. La classe Builder ha un metodo per ogni attributo dell'oggetto, che serve a impostare il valore di quell'attributo. In questo modo, si può impostare l'oggetto passo per passo, senza dover conoscere tutti i dettagli di costruzione fin dall'inizio.

Il vantaggio principale del pattern Builder è che consente di creare oggetti complessi in modo flessibile, senza dover conoscere tutti i dettagli di costruzione fin dall'inizio. Inoltre, il pattern Builder consente di creare diverse varianti dell'oggetto in modo semplice, aggiungendo semplicemente nuovi metodi alla classe Builder.

Un esempio di utilizzo del pattern Builder potrebbe essere la creazione di un oggetto di tipo Automobile, che ha molti attributi, come il modello, la marca, il numero di porte, il tipo di motore e così via. Utilizzando il pattern Builder, si potrebbe creare una classe BuilderAutomobile che ha un metodo per ogni attributo dell'automobile, e che consente di impostare l'automobile passo per passo. In questo modo, si potrebbe costruire un'automobile personalizzata, passo per passo, senza dover conoscere tutti i dettagli di costruzione fin dall'inizio.

FACTORY METHOD

Il Factory Method è un design pattern creazionale che fornisce un'interfaccia per creare oggetti in un superclasse, ma consente alle sottoclassi di modificare il tipo degli oggetti che verranno creati.

Il pattern Factory Method si basa sulla definizione di un'interfaccia comune per la creazione di oggetti e sullo spostamento della responsabilità di creazione dell'oggetto dalle classi che lo utilizzano alle classi che lo creano.

In questo modo, le classi che utilizzano gli oggetti non devono preoccuparsi della loro creazione e possono lavorare con qualsiasi oggetto che implementa l'interfaccia comune, senza conoscerne la specifica classe di implementazione.

Il Factory Method viene utilizzato quando:

- si desidera rendere il codice più flessibile e modulare, separando la creazione degli oggetti dalla loro utilizzazione;
- si desidera garantire che le classi utilizzino solo oggetti creati in modo appropriato e consistente;
- si desidera consentire alle sottoclassi di modificare la classe di oggetti che vengono creati senza dover modificare il codice delle classi che li utilizzano.

Il pattern Factory Method viene spesso utilizzato in combinazione con altri pattern, come ad esempio il Singleton e l'Abstract Factory, per fornire ancora maggiore flessibilità e controllo sulla creazione degli oggetti.

PROTOTYPE

Il design pattern Prototype è un pattern creazionale che permette di creare oggetti duplicati senza doverli istanziare direttamente. In altre parole, il pattern Prototype consente di creare nuovi oggetti utilizzando una copia di un oggetto esistente come prototipo.

In questo modo, il pattern Prototype aiuta a evitare la duplicazione di codice e riduce la complessità del codice, poiché l'istanziazione di nuovi oggetti viene gestita dal prototipo.

Il Prototype pattern viene utilizzato soprattutto quando l'istanziazione di un oggetto è costosa in termini di prestazioni e si vuole evitare di creare una nuova istanza ogni volta che viene richiesta.

Il pattern Prototype prevede la creazione di una classe prototipo, che definisce un metodo `clone` che viene utilizzato per creare una copia dell'oggetto. Inoltre, il pattern prevede l'utilizzo di una classe factory che si occupa di creare nuovi oggetti, utilizzando il prototipo come modello.

Il pattern Prototype è particolarmente utile quando si devono creare oggetti complessi che richiedono molte operazioni di inizializzazione, come ad esempio oggetti che rappresentano un database, una connessione di rete o un documento XML. Utilizzando il pattern Prototype, è possibile creare copie di questi oggetti senza dover ripetere tutte le operazioni di inizializzazione.

Il pattern Prototype è uno dei design pattern più utilizzati nella programmazione orientata agli oggetti ed è ampiamente supportato da molti linguaggi di programmazione, tra cui Java.

SINGLETON

Il design pattern Singleton è un pattern creazionale che assicura che esista una sola istanza di una classe e fornisce un punto di accesso globale a tale istanza. Questo pattern è molto utile quando abbiamo bisogno di un'istanza di una classe che deve essere condivisa in tutto il nostro programma e deve essere accessibile da diverse parti del codice.

Ecco come implementare il pattern Singleton in Java:

1. Creare una classe Singleton con un costruttore privato per impedire l'istanziazione della classe da parte di codice esterno.
2. Aggiungere una variabile statica privata alla classe Singleton che conterrà l'unica istanza della classe.
3. Fornire un metodo statico pubblico per accedere all'unica istanza della classe. Questo metodo verificherà se l'istanza esiste già e, se sì, la restituirà. Se l'istanza non esiste ancora, creerà una nuova istanza della classe e la restituirà.

Ecco un esempio di come potrebbe apparire la classe Singleton in Java:

```
public class Singleton {  
    private static Singleton instance = null;  
  
    // Costruttore privato per impedire l'istanziazione da parte di codice esterno  
    private Singleton() {  
        // ...  
    }  
  
    // Metodo statico pubblico per accedere all'unica istanza della classe  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
  
    // ...  
}
```

In questo esempio, il metodo `getInstance()` viene utilizzato per ottenere l'unica istanza della classe Singleton. Se l'istanza non esiste ancora, viene creata e restituita. In questo modo, ci assicuriamo che esista una sola istanza della classe Singleton in tutto il nostro programma.

ADAPTER

Il design pattern Adapter è un pattern strutturale che consente di convertire l'interfaccia di una classe in un'altra interfaccia che il client si aspetta di trovare. Questo pattern viene utilizzato quando una classe esistente non soddisfa le esigenze di un client a causa delle differenze tra le interfacce delle classi coinvolte.

Ecco come implementare il pattern Adapter in Java:

1. Creare un'interfaccia Target che rappresenta l'interfaccia desiderata dal client.
2. Creare una classe Adapter che implementa l'interfaccia Target e che si interfaccia con l'oggetto Adaptee.
3. Creare una classe Adaptee che rappresenta l'oggetto esistente e che non soddisfa l'interfaccia desiderata dal client.
4. Nel costruttore dell'Adapter, passare un'istanza dell'Adaptee e utilizzarla per implementare i metodi dell'interfaccia Target.

Ecco un esempio di come potrebbe apparire il pattern Adapter in Java:

```
// Interfaccia Target
public interface MediaPlayer {
    public void play(String audioType, String fileName);
}
```

```
// Classe Adaptee
public class AdvancedMediaPlayer {
    public void playVlc(String fileName) {
        // ...
    }

    public void playMp4(String fileName) {
        // ...
    }
}
```

```
// Classe Adapter
public class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedMediaPlayer;

    public MediaAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMediaPlayer = new AdvancedMediaPlayer();
        }
    }

    public void play(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedMediaPlayer.playVlc(fileName);
        }
    }
}
```

```
public class AudioPlayer implements MediaPlayer {  
    MediaAdapter mediaAdapter;  
  
    public void play(String audioType, String fileName) {  
        if (audioType.equalsIgnoreCase("mp3")) {  
            // ...  
        } else if (audioType.equalsIgnoreCase("vlc")) {  
            mediaAdapter = new MediaAdapter(audioType);  
            mediaAdapter.play(audioType, fileName);  
        } else if (audioType.equalsIgnoreCase("mp4")) {  
            // ...  
        }  
    }  
}
```

```
// Client  
public class AudioPlayer implements MediaPlayer {  
    MediaAdapter mediaAdapter;  
  
    public void play(String audioType, String fileName) {  
        if (audioType.equalsIgnoreCase("mp3")) {  
            // ...  
        } else if (audioType.equalsIgnoreCase("vlc")) {  
            mediaAdapter = new MediaAdapter(audioType);  
            mediaAdapter.play(audioType, fileName);  
        } else if (audioType.equalsIgnoreCase("mp4")) {  
            // ...  
        }  
    }  
}
```

In questo esempio, la classe AudioPlayer rappresenta il client che desidera riprodurre file audio in vari formati. Tuttavia, il client si aspetta che il metodo play riceva solo il nome del file e il tipo di audio, mentre la classe AdvancedMediaPlayer supporta solo due tipi di file audio specifici (VLC e MP4).

Per risolvere questo problema, viene utilizzata la classe MediaAdapter, che implementa l'interfaccia MediaPlayer e si interfaccia con l'oggetto AdvancedMediaPlayer. In questo modo, il client può utilizzare la classe MediaAdapter per riprodurre i file audio in formato VLC, senza dover conoscere i dettagli dell'implementazione dell'oggetto AdvancedMediaPlayer.

COMPOSITE

Il design pattern Composite è un pattern strutturale che permette di trattare gli oggetti singoli e gli oggetti composti allo stesso modo, cioè come se fossero gli stessi tipi di oggetti. In questo modo, si può creare una struttura ad albero di oggetti e accedere a ciascun oggetto in modo uniforme.

Ecco come implementare il pattern Composite in Java:

1. Creare un'interfaccia o una classe astratta Component che rappresenta gli oggetti singoli e gli oggetti composti. Questa classe o interfaccia deve fornire i metodi comuni per tutti gli oggetti.
2. Creare una classe Leaf che rappresenta gli oggetti singoli.
3. Creare una classe Composite che rappresenta gli oggetti composti. Questa classe deve contenere una lista di Component e implementare i metodi dell'interfaccia o classe astratta Component per trattare gli oggetti singoli e gli oggetti composti allo stesso modo.
4. Creare un client che costruisce la struttura ad albero di oggetti e accede ad essi tramite i metodi dell'interfaccia o classe astratta Component.

Ecco un esempio di come potrebbe apparire il pattern Composite in Java:

DECORATOR

Il design pattern Decorator è un pattern strutturale che permette di aggiungere comportamenti dinamicamente ad un oggetto senza doverlo modificare direttamente. In questo modo, si può estendere la funzionalità di un oggetto in modo flessibile e senza dover creare sottoclassi.

Ecco come implementare il pattern Decorator in Java:

1. Creare un'interfaccia o una classe astratta Component che rappresenta l'oggetto base a cui si vuole aggiungere funzionalità.
2. Creare una classe ConcreteComponent che implementa l'interfaccia o la classe astratta Component.
3. Creare una classe Decorator astratta che implementa l'interfaccia o la classe astratta Component e contiene un riferimento ad un oggetto Component. Questa classe deve implementare i metodi dell'interfaccia o classe astratta Component e delegare l'esecuzione al Component interno.
4. Creare una o più classi ConcreteDecorator che estendono il Decorator astratto e aggiungono funzionalità specifiche.

Ecco un esempio di come potrebbe apparire il pattern Decorator in Java:

```
// Interfaccia Component
public interface Pizza {
    public String getDescription();
    public double getCost();
}
```

```
// Classe ConcreteComponent
public class Margherita implements Pizza {
    public String getDescription() {
        return "Margherita";
    }

    public double getCost() {
        return 5.0;
    }
}
```

```
// Classe ConcreteDecorator
public class TomatoSauce extends ToppingDecorator {
    public TomatoSauce(Pizza pizza) {
        super(pizza);
    }

    public String getDescription() {
        return pizza.getDescription() + ", Tomato Sauce";
    }

    public double getCost() {
        return pizza.getCost() + 1.0;
    }
}
```

```
// Client
public class Client {
    public static void main(String[] args) {
        Pizza pizza = new Margherita();
        pizza = new TomatoSauce(pizza);

        System.out.println(pizza.getDescription() + " costs " + pizza.getCost() + " euros.");
    }
}
```

In questo esempio, la classe Pizza rappresenta l'interfaccia comune a tutti gli oggetti base a cui si vuole aggiungere funzionalità. La classe

Margherita rappresenta l'oggetto base. La classe ToppingDecorator rappresenta il Decorator astratto che contiene un riferimento ad un oggetto Pizza. La classe TomatoSauce rappresenta il ConcreteDecorator che estende il Decorator astratto e aggiunge funzionalità specifica.

Il client può creare un oggetto Margherita e aggiungere dinamicamente una salsa di pomodoro utilizzando la classe TomatoSauce. In questo modo, il client può estendere la funzionalità dell'oggetto base senza doverlo modificare direttamente, e senza dover creare una sottoclasse per ogni combinazione di funzionalità aggiuntive.

FACADE

Il design pattern Facade è un pattern strutturale che fornisce un'interfaccia unificata semplificando l'utilizzo di un sistema complesso di classi. Il Facade nasconde la complessità di un sistema e fornisce un'interfaccia semplice per gli utenti.

Ecco come implementare il pattern Facade in Java:

1. Identificare un sistema complesso di classi che rappresenta la logica di business.
2. Creare una classe Facade che fornisce un'interfaccia semplificata per il sistema complesso.
3. La classe Facade si occuperà di creare, inizializzare e coordinare le classi del sistema complesso.
4. Gli utenti del sistema useranno solo l'interfaccia semplificata fornita dalla classe Facade, senza dover conoscere la complessità delle classi del sistema sottostante.

Ecco un esempio di come potrebbe apparire il pattern Facade in Java:

```
// Classe complessa del sistema
public class PaymentProcessor {
    public void processPayment(String paymentMethod, double amount) {
        // Logica complessa per elaborare il pagamento
    }
}
```

```

public class PaymentFacade {
    private PaymentProcessor processor;

    public PaymentFacade() {
        this.processor = new PaymentProcessor();
    }

    public void payWithCreditCard(double amount) {
        processor.processPayment("Credit Card", amount);
    }

    public void payWithPayPal(double amount) {
        processor.processPayment("PayPal", amount);
    }

    public void payWithBitcoin(double amount) {
        processor.processPayment("Bitcoin", amount);
    }
}

```

```

// Client
public class Client {
    public static void main(String[] args) {
        PaymentFacade facade = new PaymentFacade();
        facade.payWithCreditCard(100.0);
        facade.payWithPayPal(50.0);
        facade.payWithBitcoin(200.0);
    }
}

```

In questo esempio, la classe **PaymentProcessor** rappresenta un sistema complesso di classi che si occupa di elaborare i pagamenti. La classe **PaymentFacade** rappresenta il Facade che fornisce un'interfaccia semplificata per l'elaborazione dei pagamenti. La classe **Client** utilizza

l'interfaccia semplificata fornita dalla classe **PaymentFacade** per elaborare i pagamenti.

Il client non ha bisogno di conoscere la complessità del sistema sottostante. Utilizzando la classe **PaymentFacade**, il client può elaborare i pagamenti con una semplice chiamata di metodo, senza dover conoscere la logica complessa del sistema sottostante.

PROXY

Il design pattern Proxy in Java è un pattern strutturale che fornisce un oggetto intermediario per controllare l'accesso ad un altro oggetto. Il Proxy viene utilizzato quando un'oggetto ha un costo elevato di creazione o un accesso remoto e l'utente del sistema ha bisogno di accedervi in modo trasparente e sicuro.

Ecco come implementare il pattern Proxy in Java:

1. Identificare un oggetto complesso o costoso da creare o accessibile solo in modo remoto.
2. Creare una interfaccia che rappresenta l'oggetto complesso.
3. Creare una classe Proxy che implementa l'interfaccia e ha un riferimento all'oggetto complesso.
4. La classe Proxy si occupa di controllare l'accesso all'oggetto complesso, ad esempio verificando l'autenticazione dell'utente o limitando il numero di chiamate.
5. La classe Proxy deve implementare tutti i metodi dell'interfaccia dell'oggetto complesso, delegando le richieste all'oggetto complesso solo quando necessario.

Ecco un esempio di come potrebbe apparire il pattern Proxy in Java:

```
// Interfaccia dell'oggetto complesso
public interface Image {
    void display();
}
```

```
// Classe concreta dell'oggetto complesso
public class RealImage implements Image {
    private String fileName;

    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + fileName);
    }

    private void loadFromDisk(String fileName) {
        System.out.println("Loading " + fileName);
    }
}
```

```

// Classe Proxy
public class ImageProxy implements Image {
    private RealImage image;
    private String fileName;

    public ImageProxy(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if (image == null) {
            image = new RealImage(fileName);
        }
        image.display();
    }
}

```

```

// Client
public class Client {
    public static void main(String[] args) {
        Image image1 = new ImageProxy("image1.jpg");
        Image image2 = new RealImage("image2.jpg");

        // L'immagine 1 verrà caricata solo al momento del display
        image1.display();

        // L'immagine 2 verrà caricata subito e mostrata
        image2.display();
    }
}

```

In questo esempio, la classe **RealImage** rappresenta l'oggetto complesso, ovvero un'immagine che viene caricata dal disco. La classe **ImageProxy**

rappresenta il Proxy che controlla l'accesso all'oggetto complesso, ovvero carica l'immagine solo al momento del display. La classe **Client** utilizza sia l'immagine **image1** (tramite il Proxy) che l'immagine **image2** (senza l'uso del Proxy).

In questo modo, il Proxy permette di avere un controllo più preciso sull'accesso all'oggetto complesso, senza dover caricare l'immagine immediatamente e potenzialmente sprezzando risorse.

OBSERVER

Il design pattern Observer in Java è un pattern comportamentale che consente di definire una dipendenza uno-a-molti tra gli oggetti, in modo tale che quando un oggetto cambia il suo stato, tutti gli oggetti dipendenti vengono notificati e aggiornati automaticamente.

Ecco come implementare il pattern Observer in Java:

1. Definire un'interfaccia che definisca il metodo di aggiornamento che verrà chiamato quando lo stato dell'oggetto cambia.
2. Creare una classe concreta che implementi l'interfaccia Observer e definisca il comportamento di aggiornamento.
3. Creare una classe Subject che ha una lista di Observer registrati e definisca i metodi per aggiungere, rimuovere e notificare gli Observer.
4. La classe Subject deve mantenere lo stato che verrà monitorato dagli Observer e notificare gli Observer quando lo stato cambia.
5. Ogni Observer registrato viene notificato automaticamente quando lo stato del Subject cambia.

Ecco un esempio di come potrebbe apparire il pattern Observer in Java:

```
// Interfaccia Observer
public interface Observer {
    void update(int value);
}

// Classe concreta Observer
public class ConcreteObserver implements Observer {
    @Override
    public void update(int value) {
        System.out.println("Value changed to " + value);
    }
}
```

```
// Client
public class Client {
    public static void main(String[] args) {
        Subject subject = new Subject();
        ConcreteObserver observer1 = new ConcreteObserver();
        ConcreteObserver observer2 = new ConcreteObserver();

        subject.attach(observer1);
        subject.attach(observer2);

        subject.setState(1);
        subject.setState(2);

        subject.detach(observer1);

        subject.setState(3);
    }
}
```

SCELTA DEL PATTERN APPROPRIATO

Per scegliere il design pattern giusto, è necessario identificare chiaramente il problema che si sta cercando di risolvere. Questo può comportare la valutazione di requisiti specifici o il riconoscimento di modelli ricorrenti all'interno del tuo codice. Ad esempio, se hai bisogno di creare un'istanza di una classe solo una volta per l'intera applicazione, il Singleton potrebbe essere la scelta giusta. Se hai bisogno di una modalità flessibile per creare oggetti senza esporre le loro classi concrete, potresti optare per l'Abstract Factory o il Factory Method. La scelta del design pattern dipende anche dall'architettura generale dell'applicazione. Se stai sviluppando un'applicazione web in Java, potresti considerare il pattern Model-View-Controller (MVC) per separare la logica di presentazione dalla logica di business. Se stai lavorando su un'applicazione con molte interfacce utente dinamiche, il pattern Observer potrebbe aiutarti a implementare la notifica degli eventi.

È importante bilanciare la complessità introdotta da un design pattern con i benefici che esso offre. Non è sempre necessario utilizzare un design pattern sofisticato se una soluzione più semplice è sufficiente. La scelta di un design pattern dovrebbe essere guidata dal principio KISS (Keep It Simple, Stupid), che suggerisce di preferire soluzioni più semplici quando possibile.

Infine, considera l'impatto del design pattern sulla fase di test e manutenzione del software. Assicurati che il pattern scelto semplifichi i test unitari e faciliti la manutenzione a lungo termine. Design pattern come il Dependency Injection o il Decorator possono migliorare la testabilità e la manutenibilità del codice.

GESTIONE DEI FILE IN JAVA

Introduzione

La gestione dei file rappresenta un aspetto fondamentale nell'ambito dello sviluppo software. In un linguaggio di programmazione versatile come Java, la capacità di manipolare file è cruciale per creare applicazioni che interagiscono con il sistema operativo, memorizzano dati in modo persistente e comunicano con altre applicazioni.

Java offre un insieme ricco di strumenti e librerie per la gestione dei file, che consentono agli sviluppatori di leggere, scrivere, creare, eliminare e navigare tra file e directory. Questo articolo esplorerà gli aspetti fondamentali della gestione dei file in Java, fornendo una guida pratica su come utilizzare le funzionalità offerte dal linguaggio.

La gestione dei file in Java non riguarda solo la semplice lettura e scrittura di dati, ma anche la gestione di file di configurazione, file multimediali, dati strutturati e molto altro. Attraverso l'utilizzo di concetti come stream, percorsi, operazioni di I/O, e librerie specializzate, gli sviluppatori Java possono affrontare una vasta gamma di sfide relative ai file.

Nel corso di questo articolo, esamineremo come aprire e chiudere file, leggere e scrivere dati sia in formato testo che binario, gestire le directory, e affrontare aspetti più avanzati come la gestione dei file di configurazione, la sicurezza dei file e la manipolazione di file multimediali. Sarà anche fornita una panoramica delle best practice per la gestione dei file in Java e delle risorse per ulteriori approfondimenti.

In sintesi, la gestione dei file in Java è un elemento fondamentale per gli sviluppatori che desiderano creare applicazioni robuste e versatili. Questo articolo fungerà da guida introduttiva per esplorare in dettaglio le varie

sfaccettature di questa competenza essenziale nello sviluppo software con Java.

APERTURA DI UN FILE DI TESTO

L'apertura, la lettura e la scrittura di file di testo sono operazioni comuni nella gestione dei file in Java. Vediamo come eseguire queste operazioni utilizzando le funzionalità offerte dal linguaggio.

Per aprire un file di testo in Java, è possibile utilizzare la classe `File` per rappresentare il file fisico e la classe `FileReader` per leggere il contenuto del file. Ecco un esempio:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class GestioneFileDiTesto {
    public static void main(String[] args) {
        try {
            // Specificare il percorso del file da aprire
            File file = new File("percorso_del_file.txt");

            // Creare un oggetto FileReader per leggere il file
            FileReader fileReader = new FileReader(file);

            // Ora è possibile leggere il contenuto del file
            int carattere;
            while ((carattere = fileReader.read()) != -1) {
                System.out.print((char) carattere);
            }

            // Chiudere il FileReader quando hai finito
            fileReader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Per scrivere su un file di testo, puoi utilizzare la classe `FileWriter`. Ecco un esempio di come farlo:

```
import java.io.FileWriter;
import java.io.IOException;

public class ScritturaFileDiTesto {
    public static void main(String[] args) {
        try {
            // Specificare il percorso del file in cui scrivere
            FileWriter fileWriter = new FileWriter("percorso_del_file.txt");

            // Scrivere dati nel file
            fileWriter.write("Questo è un esempio di scrittura su file di testo in Java.");
            fileWriter.write("\nAggiungiamo una nuova riga.");

            // Chiudere il FileWriter quando hai finito
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In entrambi gli esempi, è importante gestire le eccezioni per garantire che il codice funzioni correttamente e che le risorse siano liberate correttamente.

Ricorda che devi sostituire "percorso_del_file.txt" con il percorso effettivo del file di testo su cui desideri operare. Inoltre, è possibile utilizzare le classi BufferedReader e BufferedWriter per migliorare le prestazioni quando si leggono e scrivono file di testo più grandi, poiché queste classi utilizzano un buffer per le operazioni di I/O.

RANDOM ACCESS FILE

In Java, puoi creare un file ad accesso casuale utilizzando la classe RandomAccessFile. Questa classe consente di leggere e scrivere dati in un file in modo casuale, ovvero di posizionarsi in qualsiasi punto del file per effettuare operazioni di lettura o scrittura. Ecco come puoi crearlo:

```
import java.io.RandomAccessFile;
import java.io.IOException;

new *
public class CreazioneFileAccessoCasuale {
    new *
    public static void main(String[] args) {
        try {
            // Specifica il percorso del file e la modalità di apertura ("rw" per lettura e scrittura)
            RandomAccessFile file = new RandomAccessFile("file_accesso_casuale.dat", "rw");

            // Scrivi dati nel file ad accesso casuale
            String testo = "Questo è un esempio di scrittura ad accesso casuale.";
            file.writeUTF(testo);

            // Leggi dati dal file ad accesso casuale
            file.seek(0); // Posizionati all'inizio del file
            String letto = file.readUTF();
            System.out.println("Dati letti dal file: " + letto);

            // Chiudi il file
            file.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In questo esempio:

1. Viene creato un oggetto RandomAccessFile specificando il percorso del file ("file_accesso_casuale.dat") e la modalità di apertura ("rw" per lettura e scrittura).
2. Viene scritto il testo nel file utilizzando il metodo writeUTF.
3. Successivamente, viene posizionato il puntatore di lettura/scrittura all'inizio del file utilizzando il metodo seek(0).
4. Infine, il testo viene letto dal file utilizzando il metodo readUTF.

5. Assicurati di gestire le eccezioni per garantire che il codice funzioni correttamente e che il file venga chiuso in modo sicuro alla fine delle operazioni.

FILE DI PROPERTIES

file di proprietà (o file "properties") sono un formato comune per memorizzare configurazioni e impostazioni in applicazioni Java. Questi file contengono coppie chiave-valore che consentono di configurare l'applicazione senza dover modificare direttamente il codice sorgente. I file di proprietà sono spesso utilizzati per definire parametri come indirizzi URL di database, percorsi dei file, impostazioni di connessione e altre variabili di configurazione.

Ecco come si gestiscono i file di proprietà in Java:

Creazione di un File di Proprietà:

Puoi creare un file di proprietà utilizzando qualsiasi editor di testo e salvandolo con estensione ".properties". Ad esempio, il file "config.properties" potrebbe contenere le seguenti coppie chiave-valore:

```
db.url=jdbc:mysql://localhost:3306/database
db.username=utente
db.password=segreto
app.language=en
app.theme=light
```

LETTURA DI UN FILE DI PROPRIETÀ

Per leggere un file di proprietà in Java, puoi utilizzare la classe Properties fornita dal pacchetto java.util. Ecco un esempio di lettura da un file di proprietà:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

new *
public class LetturaFileProperties {
    new *
    public static void main(String[] args) {
        Properties prop = new Properties();

        try (FileInputStream input = new FileInputStream("config.properties")) {
            prop.load(input);

            // Leggi valori delle proprietà
            String dbUrl = prop.getProperty("db.url");
            String dbUsername = prop.getProperty("db.username");
            String dbPassword = prop.getProperty("db.password");

            System.out.println("URL del database: " + dbUrl);
            System.out.println("Username del database: " + dbUsername);
            System.out.println("Password del database: " + dbPassword);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

SCRITTURA IN UN FILE DI PROPRIETÀ

Per scrivere in un file di proprietà, puoi utilizzare il metodo `setProperty`. Ecco un esempio:

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

new *
public class ScrritturaFileProperties {
    new *
    public static void main(String[] args) {
        Properties prop = new Properties();

        // Impostazione delle proprietà
        prop.setProperty("app.version", "1.0");
        prop.setProperty("app.author", "John Doe");

        try (FileOutputStream output = new FileOutputStream("config.properties")) {
            // Salva le proprietà nel file
            prop.store(output, "File di configurazione dell'applicazione");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

I file di proprietà sono un modo conveniente per gestire configurazioni e impostazioni nelle applicazioni Java. Sono facili da creare, leggere e scrivere e possono essere utilizzati per personalizzare il comportamento dell'applicazione senza modificare il codice sorgente.

JDBC E SQL

Introduzione alla Java Database Connectivity

JDBC (Java Database Connectivity) è un'API standard di Java che consente agli sviluppatori di connettersi a database relazionali e di interagire con essi utilizzando il linguaggio di programmazione Java. JDBC offre un'interfaccia comune e coerente nell'accesso ai database, indipendentemente dal sistema di gestione del database (DBMS) sottostante.

Ecco come funziona JDBC e quali sono i suoi componenti chiave:

Componenti Chiave di JDBC:

Driver JDBC: Un driver JDBC è una libreria che consente a Java di comunicare con un DBMS specifico. Esistono quattro tipi di driver JDBC:

Tipo-1 (Driver JDBC di tipo ODBC): Traduce le chiamate JDBC in chiamate ODBC (Open Database Connectivity) per connettersi a un database.

Tipo-2 (Driver JDBC nativo parzialmente Java): Combina il driver Java con il driver nativo del DBMS.

Tipo-3 (Driver JDBC in rete): Utilizza un protocollo di rete intermedio per connettersi a un server DBMS remoto.

Tipo-4 (Driver JDBC nativo 100% Java): Completamente implementato in Java, comunica direttamente con il DBMS.

Interfaccia Connection: La connessione rappresenta una sessione attiva con il database. Puoi ottenere un'istanza di Connection utilizzando un URL di connessione e le credenziali di accesso al database.

Interfaccia Statement: L'interfaccia Statement consente di eseguire istruzioni SQL sul database. Puoi creare istanze di Statement per query di

selezione (SELECT), aggiornamento (INSERT, UPDATE, DELETE) e altre operazioni SQL.

Interfaccia ResultSet: Dopo l'esecuzione di un'istruzione SQL, i risultati vengono restituiti come oggetto ResultSet. Questo oggetto rappresenta un insieme di righe di dati risultanti da una query di selezione. Puoi navigare tra le righe e recuperare i dati da esse.

DriverManager: Il DriverManager è responsabile della gestione dei driver JDBC disponibili e della creazione di connessioni al database. Puoi registrare un driver utilizzando il DriverManager e ottenere connessioni utilizzando il metodo getConnection.

FLUSSO TIPICO DI USO DI JDBC

Caricamento del driver JDBC: Il driver JDBC specifico del DBMS deve essere caricato utilizzando la funzione `Class.forName("nome_driver")` o il meccanismo di registrazione automatica del driver.

Connessione al Database: Utilizzando `DriverManager.getConnection(url, username, password)`, ottieni una connessione al database specificando l'URL di connessione e le credenziali.

Creazione di Statement: Crea un'istanza di `Statement` o `PreparedStatement` per eseguire query SQL. Il `PreparedStatement` è preferibile quando si desidera evitare SQL injection.

Esecuzione delle Query: Esegui query di selezione o aggiornamento utilizzando il `Statement` o il `PreparedStatement`. I risultati delle query di selezione vengono restituiti sotto forma di oggetto `ResultSet`.

Elaborazione dei Risultati: Per le query di selezione, puoi iterare attraverso il `ResultSet` e recuperare i dati da esso. Per le query di aggiornamento, puoi ottenere il numero di righe interessate.

Chiusura delle Risorse: È importante chiudere correttamente tutte le connessioni, gli statement e i risultati utilizzando il metodo `close()` per evitare perdite di risorse.

Gestione delle Eccezioni: JDBC può generare eccezioni in caso di errori durante le operazioni di database. È fondamentale gestire queste eccezioni in modo appropriato per garantire la robustezza dell'applicazione.

JDBC fornisce un modo potente per connettersi a database, eseguire query SQL e gestire i dati in modo efficiente utilizzando il linguaggio di programmazione Java. È uno strumento essenziale per le applicazioni che necessitano di memorizzare o recuperare dati da database relazionali.

STATEMENT E PREPAREDSTATEMENT

Le interfacce Statement e PreparedStatement in JDBC consentono di eseguire istruzioni SQL su un database, ma differiscono in termini di prestazioni, sicurezza e utilizzo. Ecco una dettagliata spiegazione di entrambe:

Statement:

- Statement è un'interfaccia JDBC che consente di eseguire istruzioni SQL direttamente nel database.
- Le istruzioni SQL vengono concatenate nella stringa e inviate al database senza alcuna pre-elaborazione.
- L'uso di Statement può rendere l'applicazione vulnerabile a SQL injection, in quanto i dati inseriti direttamente nelle query possono essere manipolati in modo dannoso da utenti malevoli.

Esempio di utilizzo di Statement:

```
Statement statement = connection.createStatement();
String nome = "Alice";
String cognome = "Smith";
String query = "INSERT INTO utenti (nome, cognome) VALUES ('" + nome + "', '" + cognome + "')";
int righeModificate = statement.executeUpdate(query);
```

PREPAREDSTATEMENT

- PreparedStatement è un'interfaccia JDBC che consente di eseguire istruzioni SQL precompilate.
- Le istruzioni SQL vengono pre-elaborate dal database e ottimizzate una sola volta, riducendo il carico sul database e migliorando le prestazioni.
- I parametri nelle istruzioni SQL vengono rappresentati da segnaposto ("?") e impostati in modo sicuro tramite i metodi setTipoDato(index, valore) senza il rischio di SQL injection.
- PreparedStatement è una scelta migliore quando si lavora con query parametriche o quando si desidera evitare problemi di sicurezza legati a SQL injection.

Esempio di utilizzo di PreparedStatement:

```
String nome = "Alice";
String cognome = "Smith";
String query = "INSERT INTO utenti (nome, cognome) VALUES (?, ?)";
PreparedStatement preparedStatement = connection.prepareStatement(query);
preparedStatement.setString(1, nome);
preparedStatement.setString(2, cognome);
int righeModificate = preparedStatement.executeUpdate();
```

In sintesi, PreparedStatement è preferibile quando si eseguono query SQL con parametri o quando la sicurezza è una preoccupazione. Utilizzando i segnaposto e i metodi setTipoDato, è possibile evitare SQL injection.

D'altra parte, Statement può essere utilizzato per query non parametriche, ma è meno sicuro e può influire negativamente sulle prestazioni del database. La scelta tra Statement e PreparedStatement dipende dalle esigenze specifiche dell'applicazione.

OGGETTO RESULTSET

Un oggetto ResultSet in Java rappresenta un insieme di risultati ottenuti da un'interrogazione (query) eseguita su un database mediante JDBC (Java Database Connectivity). Il ResultSet consente di iterare attraverso i risultati e recuperare i dati dal database. Ecco come funziona il ResultSet:

Esecuzione della Query: Per ottenere un ResultSet, devi prima eseguire un'istruzione SQL, di solito tramite un oggetto Statement o PreparedStatement. L'istruzione può essere una query di selezione (SELECT) o qualsiasi altra istruzione SQL che restituisce un insieme di risultati.

```
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT nome, cognome FROM utenti");
```

Iterazione sui Risultati: Una volta ottenuto un ResultSet, puoi iterare attraverso le righe di risultati utilizzando metodi come next(). Questo metodo sposta il cursore del risultato alla successiva riga disponibile e restituisce true finché ci sono altre righe da leggere.

```
while (resultSet.next()) {
    // Esegui operazioni sui dati della riga corrente
}
```

Recupero dei Dati: Puoi recuperare i dati dalla riga corrente del ResultSet utilizzando metodi come getTipoDato("nomeColonna"). Devi specificare il tipo di dati atteso, ad esempio getString, getInt, ecc.

```
String nome = resultSet.getString("nome");
int id = resultSet.getInt("id");
```

Chiusura del ResultSet: Dopo aver completato l'iterazione e il recupero dei dati, è importante chiudere correttamente il ResultSet utilizzando il

metodo close() per rilasciare le risorse.

```
resultSet.close();
```

Gestione delle Eccezioni: È importante gestire le eccezioni quando si lavora con ResultSet. Possono verificarsi errori durante la query o problemi di connessione al database. Assicurati di utilizzare blocchi try-catch per la gestione delle eccezioni.

```
try {
    // Esecuzione della query e lavoro con il ResultSet
} catch (SQLException e) {
    e.printStackTrace();
}
```

Il ResultSet è un'interfaccia estremamente utile per recuperare dati dal database in modo dinamico. È possibile eseguire query complesse, recuperare i dati in vari formati e utilizzarli nelle applicazioni Java. È importante notare che la connessione al database, il Statement e il ResultSet devono essere correttamente chiusi per evitare perdite di risorse.

LE TRANSAZIONI

Le transazioni in JDBC (Java Database Connectivity) consentono di gestire gruppi di operazioni SQL come un'unica transazione atomica, garantendo che tutte le operazioni vengano eseguite con successo o che nessuna di esse venga eseguita. In caso di errore, è possibile annullare tutte le operazioni e ripristinare lo stato precedente del database. Le transazioni sono essenziali per garantire la coerenza dei dati in un database. Ecco come funzionano le transazioni con JDBC:

Inizio di una Transazione: Per iniziare una transazione, devi impostare l'auto-commit su false sulla connessione al database. Di solito, l'auto-commit è attivo per impostazione predefinita, il che significa che ogni istruzione SQL viene eseguita come una transazione separata. Disabilitando l'auto-commit, puoi iniziare una transazione esplicita.

```
connection.setAutoCommit(false);
```

Esecuzione delle Operazioni SQL: Durante la transazione, esegui le operazioni SQL come di consueto utilizzando gli oggetti Statement o PreparedStatement. Queste operazioni possono includere query di selezione o aggiornamento.

Commit o Rollback della Transazione: Dopo aver eseguito le operazioni, hai due opzioni:

Commit: Se tutte le operazioni hanno avuto successo e la transazione deve essere confermata, chiavi commit() sulla connessione. Questo renderà permanente l'effetto delle operazioni nel database.

```
connection.commit();
```

Rollback: Se si verifica un errore o se si desidera annullare tutte le operazioni eseguite durante la transazione, chiavi rollback() sulla connessione. Questo riporterà il database allo stato precedente alla transazione.

Riattivazione dell'Auto-Commit: Dopo il commit o il rollback, riattiva l'auto-commit sulla connessione per tornare alla modalità predefinita di esecuzione di singole operazioni come transazioni separate, questa operazione non è sempre necessaria.

```
connection.setAutoCommit(true);
```

Gestione delle Eccezioni: È importante utilizzare blocchi try-catch per gestire le eccezioni durante le operazioni SQL all'interno della transazione. In caso di eccezione, dovresti eseguire il rollback e gestire l'errore in modo appropriato.

```
try {
    // Esegui operazioni SQL all'interno della transazione
    connection.commit(); // Se tutto va bene
} catch (SQLException e) {
    connection.rollback(); // In caso di errore
    e.printStackTrace();
}
```

Questo approccio consente di garantire che un gruppo di operazioni SQL venga eseguito come una singola unità transazionale. Le transazioni sono fondamentali per garantire la consistenza e l'integrità dei dati in un database, specialmente in situazioni in cui è necessario che più operazioni siano atomiche.

BEST PRACTICE CON JDBC

- Utilizzo di PreparedStatement: Preferisci sempre l'uso di PreparedStatement rispetto a Statement quando esegui query SQL con parametri. I PreparedStatement riducono il rischio di SQL injection e migliorano le prestazioni.
- Gestione delle Eccezioni: Gestisci le eccezioni in modo adeguato. Assicurati di catturare le eccezioni e di rilasciare correttamente le risorse, come connessioni, statement e risultati.
- Chiusura delle Risorse: Sempre chiudi esplicitamente le risorse (connessioni, statement, risultati) utilizzando i metodi close(). Usa blocchi try-with-resources o blocchi finally per garantire la chiusura delle risorse anche in caso di eccezioni.
- Transazioni: Usa transazioni per garantire che un gruppo di operazioni SQL venga eseguito come un'unica unità atomica. Inizia la transazione con connection.setAutoCommit(false), esegui le operazioni e, in caso di successo, conferma con connection.commit(), altrimenti esegui il rollback con connection.rollback().
- Gestione delle Connessioni: Usa un pool di connessioni per gestire le connessioni al database in modo efficiente. Framework come Apache Commons DBCP o HikariCP forniscono pool di connessioni pronti per l'uso.
- Caricamento del Driver JDBC: Carica il driver JDBC una sola volta all'avvio dell'applicazione utilizzando Class.forName() o il meccanismo di registrazione automatica del driver.
- Utilizzo di try-catch con le Risorse: Utilizza blocchi try-with-resources per gestire le chiusure delle risorse in modo automatico. Ad esempio:

```
try (Connection connection = dataSource.getConnection();
      PreparedStatement preparedStatement = connection.prepareStatement(sql);
      ResultSet resultSet = preparedStatement.executeQuery()) {
    // Lavoro con il ResultSet
} catch (SQLException e) {
    e.printStackTrace();
}
```

- Preferisci il Framework ORM: Quando possibile, considera l'uso di framework ORM (Object-Relational Mapping) come Hibernate o JPA (Java Persistence API) per semplificare l'interazione con il database e ridurre la complessità del codice SQL.
- Validazione dei Dati: Sempre convalida e sanifica i dati in ingresso per prevenire SQL injection. Usa metodi come PreparedStatement con segnaposto per parametri per evitare l'iniezione di SQL.
- Scalabilità: Prendi in considerazione la scalabilità dell'applicazione e la gestione delle connessioni al database. Evita di tenere aperte le connessioni per lunghi periodi e assicurati di rilasciare le connessioni non utilizzate.
- Monitoraggio e Ottimizzazione: Monitora le query SQL eseguite e ottimizza le prestazioni. Usa strumenti di profilazione per identificare le query lente e migliorale se necessario.
- Sicurezza: Proteggi le credenziali di accesso al database. Non memorizzare mai le password in chiaro nel codice sorgente o nei file di configurazione. Usa meccanismi sicuri per gestire le credenziali.
- Logging: Usa un sistema di logging per registrare le attività del database e le eccezioni. Questo è utile per il debugging e il monitoraggio.
- Testing: Esegui test unitari e di integrazione per verificare il corretto funzionamento delle operazioni di database.

Seguendo queste best practice, puoi sviluppare applicazioni JDBC robuste, sicure ed efficienti che interagiscono con database relazionali in modo affidabile.

ROBLEMI DI SICUREZZA SQL INJECTION

SQL Injection è una tecnica di attacco informatico in cui un aggressore inserisce deliberatamente istruzioni SQL dannose o malevoli all'interno delle viste di input dei campi dei form, dei parametri delle query o dei filtri in un'applicazione web. Queste istruzioni SQL dannose vengono quindi eseguite dal database, compromettendo la sicurezza dell'applicazione e consentendo all'attaccante di accedere o manipolare dati sensibili o danneggiare il database stesso. Ecco come funziona SQL Injection e come evitarlo:

COME FUNZIONA SQL INJECTION

L'applicazione riceve dati in ingresso dall'utente, come parametri di una query SQL o dati da un modulo web.

Un aggressore inserisce dati dannosi o malevoli in uno o più campi di input. Ad esempio, potrebbe inserire il seguente testo in un campo username:

```
' OR '1'='1
```

L'applicazione concatena questi dati direttamente in un'istruzione SQL senza sanitizzare o convalidare i dati in modo appropriato.

L'istruzione SQL risultante diventa qualcosa del genere:

```
SELECT * FROM utenti WHERE username = '' OR '1'='1' AND password = 'password123'
```

L'istruzione SQL viene eseguita dal database, restituendo dati non autorizzati.

Come Evitare SQL Injection:

Per prevenire SQL Injection, è fondamentale seguire queste pratiche di sicurezza:

Parametrizzare le Query: Utilizza PreparedStatement o query parametriche per separare chiaramente i dati da inserire nella query. Questo consente al driver JDBC di gestire i dati in modo sicuro. Ad esempio:

```
String query = "SELECT * FROM utenti WHERE username = ? AND password = ?";  
PreparedStatement preparedStatement = connection.prepareStatement(query);  
preparedStatement.setString(1, username);  
preparedStatement.setString(2, password);  
ResultSet resultSet = preparedStatement.executeQuery();
```

- Convalida e Sanifica i Dati: Prima di utilizzare i dati in un'istruzione SQL, convalidali e sanificali. Assicurati che i dati soddisfino i requisiti attesi e rimuovi eventuali caratteri dannosi. Utilizza librerie di convalida, come Apache Commons Validator, per semplificare questo processo.
- Evita la Concatenazione Manuale: Evita di concatenare manualmente i dati utente nelle query SQL. È sempre preferibile utilizzare query parametriche o metodi specifici per aggiungere dati ai comandi SQL.
- Autorizzazioni di Accesso: Limita l'accesso del database dell'applicazione alle sole autorizzazioni necessarie. Non utilizzare account con autorizzazioni di amministratore per l'applicazione.
- Gestione delle Eccezioni: Cattura e gestisci le eccezioni in modo appropriato. Non esporre informazioni sensibili sugli errori all'utente finale.
- Aggiornamenti e Patch: Mantieni il tuo software e i componenti di terze parti aggiornati con patch di sicurezza per proteggerti da vulnerabilità note.
- Auditing e Monitoraggio: Implementa un sistema di auditing e monitoraggio per rilevare attività sospette o tentativi di SQL Injection.
- Limita l'Informazione in Errore: Non fornire dettagli dettagliati sugli errori SQL al cliente. Tieni un registro degli errori lato server per il debugging.
- Utilizzo di Framework ORM: Quando possibile, utilizza framework ORM come Hibernate o JPA, che gestiscono

automaticamente le query SQL e le proteggono da SQL Injection.

- Formazione e Consapevolezza: Forma il personale coinvolto nello sviluppo per essere consapevole delle minacce di sicurezza e delle migliori pratiche di prevenzione.

La prevenzione di SQL Injection è fondamentale per la sicurezza delle applicazioni web e dei database. Seguendo queste pratiche, puoi ridurre notevolmente il rischio di vulnerabilità legata a SQL Injection.

ESERCIZIO 13: LETTURA DI DATI DA UN DATABASE

1. Crea una tabella di esempio in un database MySQL o SQLite con alcune colonne (ad esempio, "id," "nome," "cognome," "età").
2. Scrivi un'applicazione Java che si connette al database utilizzando JDBC e legge tutti i dati dalla tabella.
3. Visualizza i dati letti nel terminale o in un'applicazione Java.

ESERCIZIO 14: INSERIMENTO DI DATI IN UN DATABASE

1. Usa lo stesso database creato nell'Esercizio 1.
2. Scrivi un'applicazione Java che consente all'utente di inserire nuovi dati (ad esempio, "nome," "cognome," "età") in una tabella del database.
3. Dopo l'inserimento, conferma che i dati siano stati aggiunti con successo eseguendo una query di lettura.

ESERCIZIO 15:

AGGIORNAMENTO E CANCELLAZIONE DI DATI

1. Ancora una volta, usa lo stesso database creato nei due esercizi precedenti.
2. Scrivi un'applicazione Java che consente all'utente di selezionare un record esistente dal database e di aggiornarne una o più colonne (ad esempio, "età").
3. Includi anche la funzionalità per cancellare un record selezionato.

I THREAD

I thread sono uno dei concetti fondamentali nella programmazione concorrente e parallela. In Java, un thread è un sottoprocesso leggero che consente a un'applicazione di eseguire più compiti contemporaneamente, migliorando l'efficienza e la reattività del software. Ecco una spiegazione di cosa sono i thread in Java e come crearli:

Cos'è un Thread in Java:

- Un thread è una singola sequenza di esecuzione all'interno di un processo Java.
- Java utilizza il modello di threading multithread, che consente l'esecuzione simultanea di più thread all'interno di un'applicazione.
- I thread condividono lo stesso spazio di memoria, consentendo loro di accedere e condividere dati.
- I thread sono utili per svolgere attività parallele, come elaborazione di dati in background, gestione di eventi, e altro.

Creazione di Thread in Java: In Java, puoi creare thread estendendo la classe Thread o implementando l'interfaccia Runnable. Ecco come farlo:

Estendere la classe Thread (Ereditarietà):

```
public class MioThread extends Thread {  
    public void run() {  
        // Il codice eseguito dal thread va qui  
    }  
  
    // Creazione e avvio del thread  
    MioThread mioThread = new MioThread();  
    mioThread.start();
```

Implementare l'interfaccia Runnable (Composizione):

```
public class MioRunnable implements Runnable {  
    public void run() {  
        // Il codice eseguito dal thread va qui  
    }  
}  
  
// Creazione di un oggetto Runnable  
MioRunnable mioRunnable = new MioRunnable();  
  
// Creazione e avvio del thread utilizzando un oggetto Runnable  
Thread mioThread = new Thread(mioRunnable);  
mioThread.start();
```

Quando si utilizza un oggetto Thread, si estende direttamente la classe Thread e si ridefinisce il metodo run(). Quando si utilizza un oggetto Runnable, si implementa l'interfaccia Runnable e si fornisce l'oggetto Runnable a un oggetto Thread.

Puoi anche utilizzare thread anonimi o lambdas per semplificare la creazione e l'avvio di thread. Ad esempio:

```
// Thread anonimo  
Thread mioThread = new Thread(() -> {  
    // Codice eseguito dal thread va qui  
});  
mioThread.start();
```

È importante notare che la gestione corretta dei thread in Java comporta la sincronizzazione e la prevenzione di condizioni di gara e problemi di concorrenza. Java fornisce meccanismi come synchronized blocks e Locks per gestire questi aspetti. Inoltre, puoi utilizzare la classe ExecutorService per gestire thread in modo più efficiente, ad esempio in un pool di thread.

LA CONCORRENZA

I problemi di concorrenza si verificano quando più thread all'interno di un'applicazione accedono o modificano le stesse risorse condivise contemporaneamente. Questi problemi possono causare comportamenti indesiderati o imprevedibili nelle applicazioni, inclusi errori, crash o risultati non deterministici. Ecco alcuni dei problemi di concorrenza più comuni e come possono essere risolti:

1. **Race Condition (Condizione di Gara):** Un race condition si verifica quando due o più thread tentano di accedere o modificare una risorsa condivisa allo stesso tempo, causando risultati imprevedibili. Per risolvere i race condition, puoi utilizzare la sincronizzazione, ad esempio utilizzando blocchi synchronized o ReentrantLock.
2. **Deadlock (Blocco dei Thread):** Un deadlock si verifica quando due o più thread si bloccano a vicenda aspettando che l'altro rilasci una risorsa. Per prevenire i deadlock, devi garantire che i thread acquisiscano le risorse nello stesso ordine in tutta l'applicazione o utilizzino timeout e strategie di rilascio delle risorse.
3. **Starvation (Inedia):** La starvation si verifica quando un thread non riesce a ottenere l'accesso a una risorsa per lungo tempo a causa di una priorità troppo bassa o di altri thread che monopolizzano le risorse. Per evitare la starvation, puoi utilizzare la pianificazione dei thread, assegnando priorità ai thread o utilizzando algoritmi di gestione delle code.
4. **Contention (Concorrenza):** La contention si verifica quando troppi thread cercano di accedere alla stessa risorsa contemporaneamente, causando un calo delle prestazioni. Per risolvere la contention, puoi utilizzare tecniche come l'accesso

condiviso limitato, il partizionamento delle risorse o la riduzione del tempo di accesso alle risorse.

5. **Inconsistenza dei Dati:** L'inconsistenza dei dati si verifica quando i dati vengono letti o scritti da più thread contemporaneamente, causando la perdita o l'alterazione dei dati. Per risolvere questo problema, puoi utilizzare la sincronizzazione e l'accesso ai dati in modo atomico o utilizzare strutture dati con supporto alla concorrenza.
6. **Memory Model: Conflitto tra Caching e Visibilità:** I moderni processori eseguono spesso ottimizzazioni di caching dei dati, che possono causare problemi di visibilità tra thread. Per risolvere questi problemi, puoi utilizzare le parole chiave volatile o synchronized per garantire la visibilità dei dati tra i thread.
7. **Fifo Starvation (Inedia FIFO):** Questo problema si verifica quando i thread vengono serviti in base a una politica FIFO e alcuni thread a bassa priorità rimangono in attesa indefinitamente. Puoi risolvere questo problema utilizzando politiche di gestione dei thread più complesse, come la priorità dinamica o l'aggiornamento delle priorità in base al tempo di attesa.

La gestione della concorrenza è una parte fondamentale della progettazione e dello sviluppo di applicazioni multithreading. La scelta delle tecniche di gestione della concorrenza dipende dalle esigenze specifiche dell'applicazione, ma in generale è importante utilizzare le strutture dati e i meccanismi di sincronizzazione forniti da Java in modo oculato per prevenire problemi di concorrenza.

ESERCIZIO 1: CONTATORE CONDIVISO

Scopri come più thread possono condividere una variabile contatore e aggiornarla in modo concorrente. Crea due thread che aumentano il contatore da 1 a 10000 ciascuno e verifica se il risultato è corretto alla fine.

ESERCIZIO 2: PRODUTTORE-CONSUMATORE

Implementa un problema di produttore-consumatore utilizzando thread. Hai un produttore che produce elementi e li inserisce in un buffer condiviso e un consumatore che preleva e consuma gli elementi dal buffer. Assicurati che il produttore non produca troppi elementi e che il consumatore non tenti di consumare quando il buffer è vuoto.

ESERCIZIO 3: LETTORI E SCRITTORI

Implementa un problema di lettori-scrrittori utilizzando thread. Hai più lettori che leggono da una risorsa condivisa e uno scrittore che scrive su questa risorsa. Assicurati che i lettori possano leggere contemporaneamente, ma quando lo scrittore scrive, nessun lettore può leggere e nessun altro scrittore può scrivere.

COMUNICAZIONE TRAMITE SOCKET IN JAVA

La comunicazione tramite socket è una delle tecnologie fondamentali nella programmazione di rete. In Java, è possibile utilizzare le classi e le API del pacchetto `java.net` per creare connessioni di rete, consentendo a due o più applicazioni di comunicare tra loro attraverso una rete. Questo capitolo esplorera come utilizzare le socket in Java per creare applicazioni di rete che possono scambiare dati tra client e server.

CONCETTI FONDAMENTALI DELLE SOCKET

Le socket sono endpoint per la comunicazione bidirezionale tra due computer su una rete. Esistono due tipi principali di socket in Java: socket client e socket server.

- **Socket Client:** Il client si connette a un server e invia richieste al server. Può essere un'applicazione che richiede dati o servizi da un server remoto.
- **Socket Server:** Il server è in ascolto su una porta specifica e attende le connessioni dai client. Quando si stabilisce una connessione, il server può leggere i dati inviati dai client e rispondere di conseguenza.

CREAZIONE DI UNA SOCKET CLIENT

Per creare una socket client in Java, puoi seguire questi passaggi:

Importa le classi necessarie:

```
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;
```

Crea una connessione ai server socket specificando l'indirizzo IP e la porta del server:

```
String serverIP = "127.0.0.1"; // Esempio: indirizzo IP del server
int serverPort = 12345; // Esempio: porta del server
Socket socket = new Socket(serverIP, serverPort);
```

Ottieni gli stream di input e output per la comunicazione:

Puoi ora inviare dati al server utilizzando l'oggetto output e ricevere dati dal server utilizzando l'oggetto input.

Alla fine, ricordati di chiudere la socket e i flussi quando hai finito:

```
socket.close();
input.close();
output.close();
```

CREAZIONE DI UNA SOCKET SERVER

Per creare una socket server in Java, puoi seguire questi passaggi:

Importa le classi necessarie:

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;
```

Crea un oggetto ServerSocket e specifica la porta su cui il server ascolterà le connessioni:

```
int serverPort = 12345; // Esempio: porta del server
ServerSocket serverSocket = new ServerSocket(serverPort);
```

Utilizza un loop per accettare le connessioni in arrivo dai client:

```
while (true) {
    Socket clientSocket = serverSocket.accept(); // Accetta la connessione dal client
    // Gestisci la connessione in un thread separato per consentire più connessioni simultanee
}
```

All'interno del loop, puoi ottenere gli stream di input e output per la comunicazione con il client:

```
DataOutputStream output = new DataOutputStream(clientSocket.getOutputStream());
DataInputStream input = new DataInputStream(clientSocket.getInputStream());
```

Puoi ora leggere dati dal client utilizzando input e inviare dati al client utilizzando output.

Alla fine, chiudi la socket del client e i flussi quando hai finito con la connessione:

```
clientSocket.close();
input.close();
output.close();
```

Continua ad accettare nuove connessioni utilizzando il loop

ESEMPIO DI APPLICAZIONE CLIENT-SERVER

Ecco un esempio semplice di un'applicazione client-server che consente al client di inviare un messaggio al server e ricevere una risposta. Nota che questa è solo una base di partenza, e puoi estendere questa applicazione per soddisfare requisiti più complessi.

Server:

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;

new *
public class Server {
    new *
    public static void main(String[] args) {
        try {
            int serverPort = 12345;
            ServerSocket serverSocket = new ServerSocket(serverPort);
            System.out.println("Server in ascolto sulla porta " + serverPort);

            while (true) {
                Socket clientSocket = serverSocket.accept();
                DataInputStream input = new DataInputStream(clientSocket.getInputStream());
                DataOutputStream output = new DataOutputStream(clientSocket.getOutputStream());

                String message = input.readUTF();
                System.out.println("Messaggio ricevuto dal client: " + message);

                String response = "Messaggio ricevuto con successo!";
                output.writeUTF(response);

                clientSocket.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Client;

```
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;

new *

public class Client {
    new *
    public static void main(String[] args) {
        try {
            String serverIP = "127.0.0.1";
            int serverPort = 12345;

            Socket socket = new Socket(serverIP, serverPort);
            DataInputStream input = new DataInputStream(socket.getInputStream());
            DataOutputStream output = new DataOutputStream(socket.getOutputStream());

            String message = "Ciao, server!";
            output.writeUTF(message);

            String response = input.readUTF();
            System.out.println("Risposta dal server: " + response);

            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

In questo esempio, il client invia un messaggio al server, il server lo riceve e invia una risposta al client. Puoi eseguire il server e il client su macchine diverse o sulla stessa macchina con porte diverse per testare la comunicazione.

La programmazione con socket in Java offre la possibilità di creare applicazioni di rete personalizzate e connettersi a server remoti. Questo capitolo ha introdotto i concetti fondamentali delle socket in Java e fornito un esempio di base di un'applicazione client-server. La comunicazione tramite socket è una tecnologia potente che può essere utilizzata per una varietà di casi d'uso, dalla messaggistica istantanea al trasferimento di dati in tempo reale su reti locali o su Interne

SISTEMA DI CHAT CLIENT SERVER CHE CONSERVA I MESSAGGI

Da questo capitolo in poi, il sistema di apprendimento non seguirà più il classico schema argomento trattato, esercizio. Seguiremo le fasi di progettazione e sviluppo di un vero applicativo, soffermandoci tutte le volte che incontreremo nuovi argomenti, l'applicativo che tratteremo ha solo uno scopo didattico, ma seguiremo egualmente tutte le fasi che portano alla sua realizzazione, partendo dai requisiti utente fino all'implementazione.

I nuovi temi trattati tuttavia non saranno approfonditi, ma serviranno ad aggiungere nuovi elementi per portare a termine il nostro progetto.

REQUISITI UTENTE

L'applicazione che useremo come caso di studio si chiama Chat-Up è un sistema di messaggistica molto semplice, consente agli utenti di inviare e ricevere messaggi in due modalità.

- **Messaggi globali:** messaggi visibili a tutti gli utenti registrati in chat
- **Messaggi privati:** in questo caso la conversazione è visibile solo tra due elementi

La registrazione alla chat non prevede nessun tipo di sicurezza, l'unica regola è che l'utente che entra in chat deve avere un nickname e questo deve essere univoco, inoltre i messaggi verranno memorizzati in un database e potranno essere visionati dagli utenti in accessi successivi.

L'applicativo ha un architettura client server, il server gestisce le richieste degli utenti che accederanno e comunicheranno con un programma client, che ha un'interfaccia utente user friendly,

il client ha una gui Windows style, composto da una area chat, dove vengono visualizzati i messaggi e un text field dove l'utente può scrivere inviare i messaggi, per ogni chat verrà creato un tab panel, il tab panel con le chat globali sarà sempre visibile, questo tab panel potrà visualizzerà una lista che contiene tutti i nickname presenti in chat.

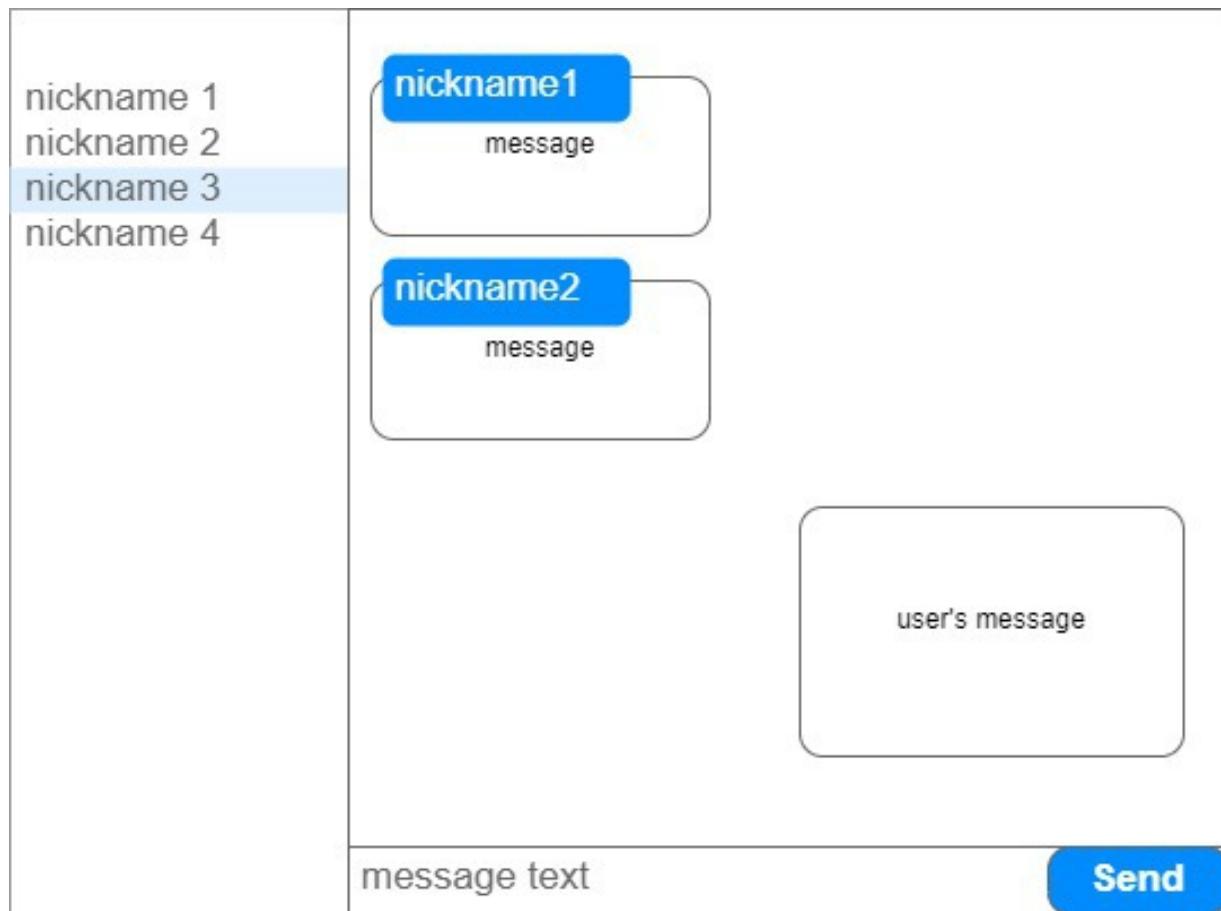


Figura 1 Mockup global chat

Dalla lista degli utenti si potrà avviare una nuova chat, facendo doppio click sul nickname dell’utente che si vuole contattare.

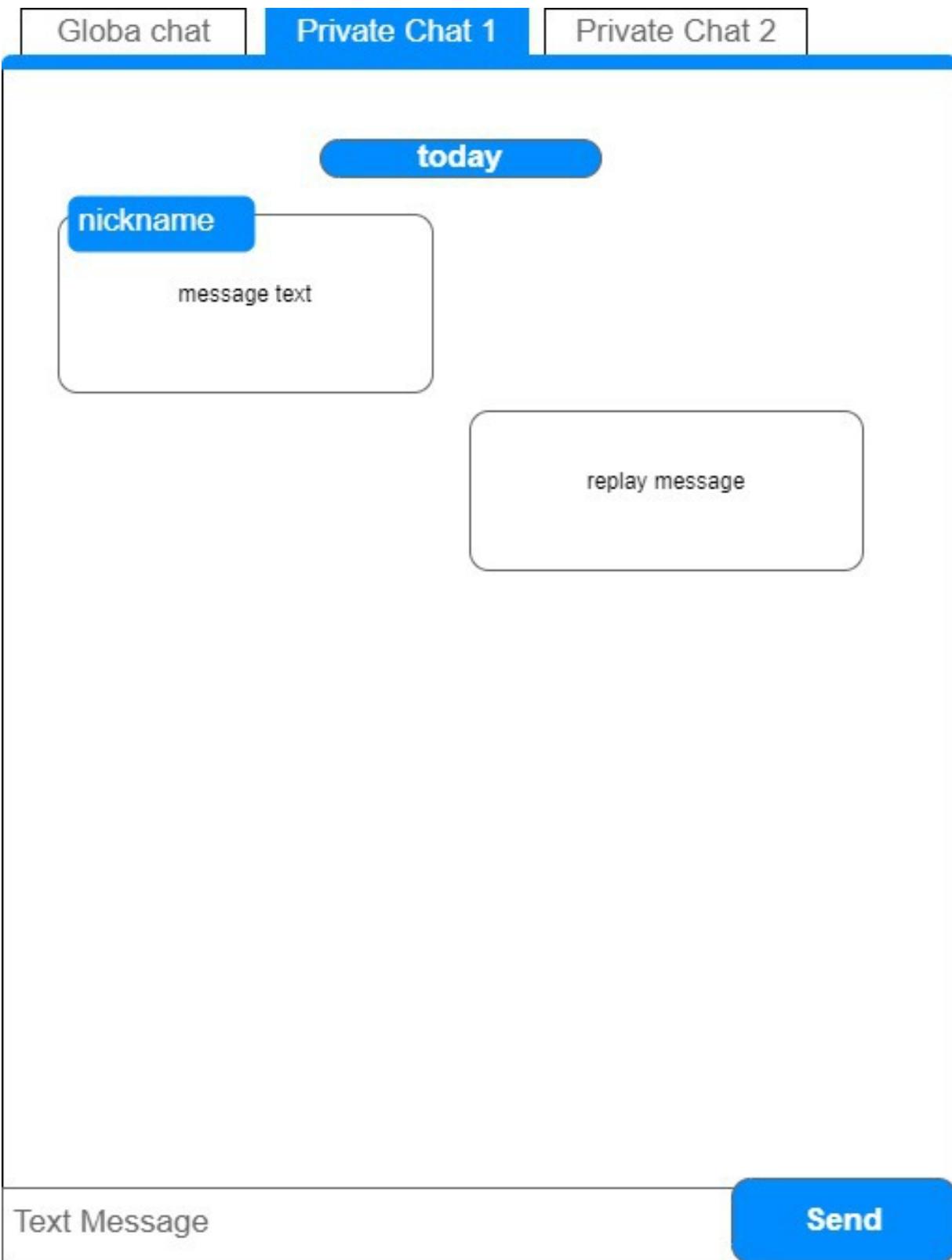


Figura 2 Mockup private chat

Verrà aperto un nuovo tab panel con la chat privata visibile solo ai due utenti collegati.

ANALISI FUNZIONALE

Per raggruppare e analizzare tutte le funzioni della nostra chat, ci faremo aiutare da un diagramma UML use case:

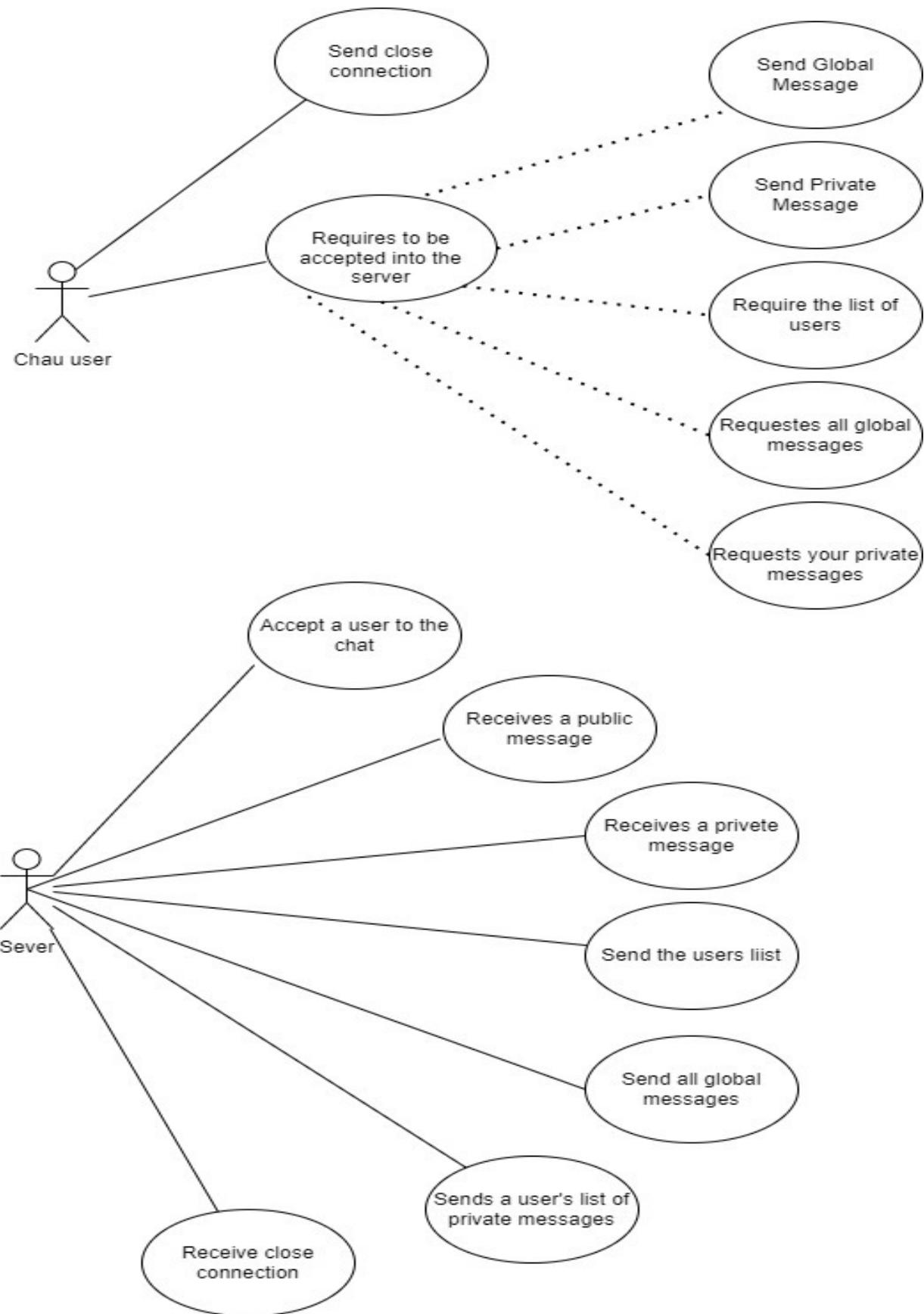


Figura 3 Uml use case Chatup application

Come possiamo vedere dal diagramma gli attori in gioco nel nostro sistema sono due, Utente della chat e il Server, che dialogheranno tra di loro tramite l'utilizzo di un protocollo di comunicazione, questo permette ai due attori di comunicare con la stessa lingua

I'utente della chat implementerà le seguenti funzionalità:

- **Richiesta di accettazione al server**, dopo aver effettuato la connessione al server ogni utente richiederà l'accettazione ad entrare nella chat, la richiesta viene fatta inviando un nickname con il quale l'utente sarà identificato nel sistema, questa operazione richiederà un solo controllo da parte del server se il nickname è già presente la richiesta verrà rifiutata, altrimenti verrà inviato come esito positivo il numero di versione del protocollo di comunicazione Client-Server. A sua volta il client verificherà che la versione del protocollo che usa il server sia la stessa con cui è stato implementato, altrimenti chiuderà la connessione.
- **Notifica di chiusura connessione**, in qualsiasi momento il client potrà notificare la chiusura della connessione al server che a sua volta notificherà a tutti gli utenti che l'utente non è più disponibile nella chat.
- **Richiesta della lista degli utenti**, dopo aver ricevuto l'accettazione del server ad entrare nella chat il client richiederà la lista degli utenti che sono attivi in quel momento.
- **Richiesta di messaggi globali**, il client richiederà al server la tutti messaggi presenti nella chat globale.
- **Richiesta dei messaggi privati**, il client richiederà al server la tutti i suoi messaggi privati.
- **Invio di un messaggio globale**, il client potrà in qualsiasi momento inviare un messaggio alla chat globale, questo implicherà che il server notificherà a tutti gli utenti che è stato inviato questo messaggio.

- **Invio di un messaggio privato**, il client in qualsiasi momento potrà inviare un messaggio privato ad un altro utente, questi implicherà che il server invierà il messaggio solo all'utente selezionato per la chat privata.
- **Ricezione messaggi**, il client resterà in attesa della ricezione di nuovi messaggi da parte del server e capirà se un messaggio nella chat globale o è un messaggio privato.

Il server implementa le seguenti funzionalità:

- **Accetta le richieste di entrare in chat**, resta in attesa delle richieste di entrare nel sistema, controlla che il nickname scelto dall'utente non sia presente, in caso contrario invia un messaggio di rifiuto.
- **Invia la lista degli utenti**, resta in attesa delle richieste di invio della lista utenti.
- **Riceve messaggi pubblici e privati**, resta in attesa di messaggi da parte degli utenti della chat, e se è un messaggio pubblico lo invia a tutti gli utenti connessi, e se è privato lo invia a solo al destinatario, inoltre i messaggi sia pubblici che privati verranno salvati su un database, per permettere agli utenti di poter leggere i messaggi anche in tempi successivi.
- **Riceve richieste di messaggi**. Il server rimane in attesa delle richieste di messaggi da parte dei client, effettua la ricerca sul database e invia i risultati.
- **Chiusura di una connessione**. Il server resta in attesa dei messaggi di chiusura della connessione da parte degli utenti, in caso di notifica invierà un messaggio a tutti client che quel utente non è più attivo.

IL PROTOCOLLO DI COMUNICAZIONE

Un protocollo di comunicazione è un insieme di regole e convenzioni che definiscono il formato, l'ordine e il comportamento delle informazioni scambiate tra dispositivi o entità all'interno di un sistema di comunicazione. Queste regole stabiliscono come i dati devono essere trasmessi, ricevuti, elaborati e interpretati durante una comunicazione.

Alcuni punti chiave relativi ai protocolli di comunicazione:

- **Standardizzazione:** I protocolli di comunicazione vengono utilizzati per standardizzare il modo in cui le informazioni vengono scambiate tra i partecipanti a una comunicazione. Questa standardizzazione è essenziale per garantire che i dispositivi o le applicazioni diverse siano in grado di comunicare tra loro in modo efficace.
- **Formato dei dati:** I protocolli definiscono il formato dei dati, inclusi gli header, i campi dei dati e, se del caso, i controlli di errore o di sicurezza. Ad esempio, il formato di un messaggio può essere testuale (ad esempio JSON, XML) o binario (ad esempio TCP/IP).
- **Struttura e sequenza:** I protocolli specificano l'ordine e la sequenza delle operazioni e delle informazioni durante una comunicazione. Questo può includere la negoziazione delle connessioni, l'inizio e la fine di una sessione di comunicazione e l'invio di richieste e risposte.
- **Gestione degli errori:** La gestione degli errori è un componente importante dei protocolli di comunicazione. Questi protocolli definiscono come gestire situazioni di errore, come la ritrasmissione di dati persi o danneggiati, la gestione degli errori di sintassi e la sicurezza.

- **Sicurezza:** Molti protocolli includono anche aspetti di sicurezza, come l'autenticazione e la crittografia, per proteggere i dati durante la trasmissione e garantire che solo le parti autorizzate abbiano accesso alle informazioni.
- **Applicazioni:** I protocolli di comunicazione sono utilizzati in una vasta gamma di applicazioni, dalla comunicazione tra computer in una rete (come il protocollo TCP/IP utilizzato su Internet) alla comunicazione tra componenti software o dispositivi hardware diversi (come il protocollo USB per periferiche).

Esempi di protocolli di comunicazione noti includono HTTP (Hypertext Transfer Protocol) utilizzato per la comunicazione Web, SMTP (Simple Mail Transfer Protocol) per l'invio di email e TCP/IP (Transmission Control Protocol/Internet Protocol) per la comunicazione su Internet. Ognuno di questi protocolli ha specifiche regole e convenzioni che definiscono come i dati vengono scambiati tra i partecipanti alla comunicazione.

Il protocollo di comunicazione della nostra chat sarà molto semplice, si basa su parole chiave che vengono interpretate come comandi, e uno o più parametri, il comando dal parametro vengono divisi da un carattere separatore, noi utilizzeremo la chiocciola '@' ecco la lista e come sono formattate queste key:

accept@<nickname> Inviato dal client al server per poter entrare nella chat, il parametro da passare è il nickname scelto dall'utente che deve essere univoco

welcome@<versione del protocollo> Inviato dal server, al client per confermare l'ingresso nella chat e comunicare la versione del protocollo usata, il client deciderà di abbandonare la connessione se la versione del protocollo è diversa.

global@<messaggio>@<nickname> Inviato dal client al server, per spedire un messaggio globale (visibile a tutti gli utenti), come secondo parametro è presente anche il nickname del mittente.

private@<messaggio>@<nickname da>@<nickname a > Inviato dal client al server per recapitare un messaggio privato ad un utente della chat, in questo caso bisognerà specificare il nickname del mittente e il nickname del destinatario.

list inviato dal client al server, per richiedere lista di tutti i nickname presenti in chat.

rcvlist@<lista degli utenti> Inviato dal server al client in seguito ad una richiesta della lista utenti, come parametro la lista degli utenti, un'unica stringa con ogni record delimitato da una lunghezza fissa.

getglobal Inviato dal client al server per richiedere la lista di tutti i messaggi pubblici.

rcvglobal@<lista messaggi pubblici> Inviato dal server al client in seguito ad una richiesta getglobal, come parametro c'è una stringa con i messaggi, ogni record è delimitato a lunghezza fissa.

getprivate Inviato dal client al server, per richiedere la lista dei messaggi privati.

rcvprivate@<lista messaggi privati> Inviato dal server al client in seguito ad una richiesta di getprivate come parametro c'è una stringa con i messaggi, ogni record è delimitato a lunghezza fissa.

close Inviato dal client al server per comunicare la chiusura della connessione.

LE ENUMERATION

Per implementare il nostro protocollo possiamo pensare di raggruppare tutti i comandi indicati in una classe enumeration.

In Java, le enumerazioni (enum) sono un tipo di dato che rappresenta un insieme fisso di costanti nome-valore. Un'enumerazione è definita utilizzando la keyword enum e può essere utilizzata per dichiarare un insieme limitato di valori che rappresentano concetti ben definiti.

Ecco un esempio semplice di come definire e utilizzare un'enumerazione in Java:

```
enum GiorniSettimana {  
    LUNEDI,  
    MARTEDI,  
    MERCOLEDI,  
    GIOVEDI,  
    VENERDI,  
    SABATO,  
    DOMENICA  
}
```

In questo modo abbiamo raggruppato i giorni della settimana in un'unica entità, possiamo poi applicare la nostra enumeration in vari modi:

```
// Utilizzo dell'enumerazione
GiorniSettimana giorno = GiorniSettimana.MARTEDI;

// Switch statement con un'enumerazione
switch (giorno) {
    case LUNEDI:
        System.out.println("Primo giorno della settimana.");
        break;
    case MARTEDI:
        System.out.println("Secondo giorno della settimana.");
        break;
    // Altri casi per gli altri giorni della settimana...
    default:
        System.out.println("Altro giorno della settimana.");
}
```

In questo esempio, abbiamo definito un'enumerazione chiamata GiorniSettimana, che rappresenta i giorni della settimana. Ogni costante all'interno dell'enumerazione rappresenta un giorno specifico.

```
// Ottenere tutti i valori dell'enumerazione
GiorniSettimana[] giorni = GiorniSettimana.values();
System.out.println("Tutti i giorni della settimana:");
for (GiorniSettimana g : giorni) {
    System.out.println(g);
}
```

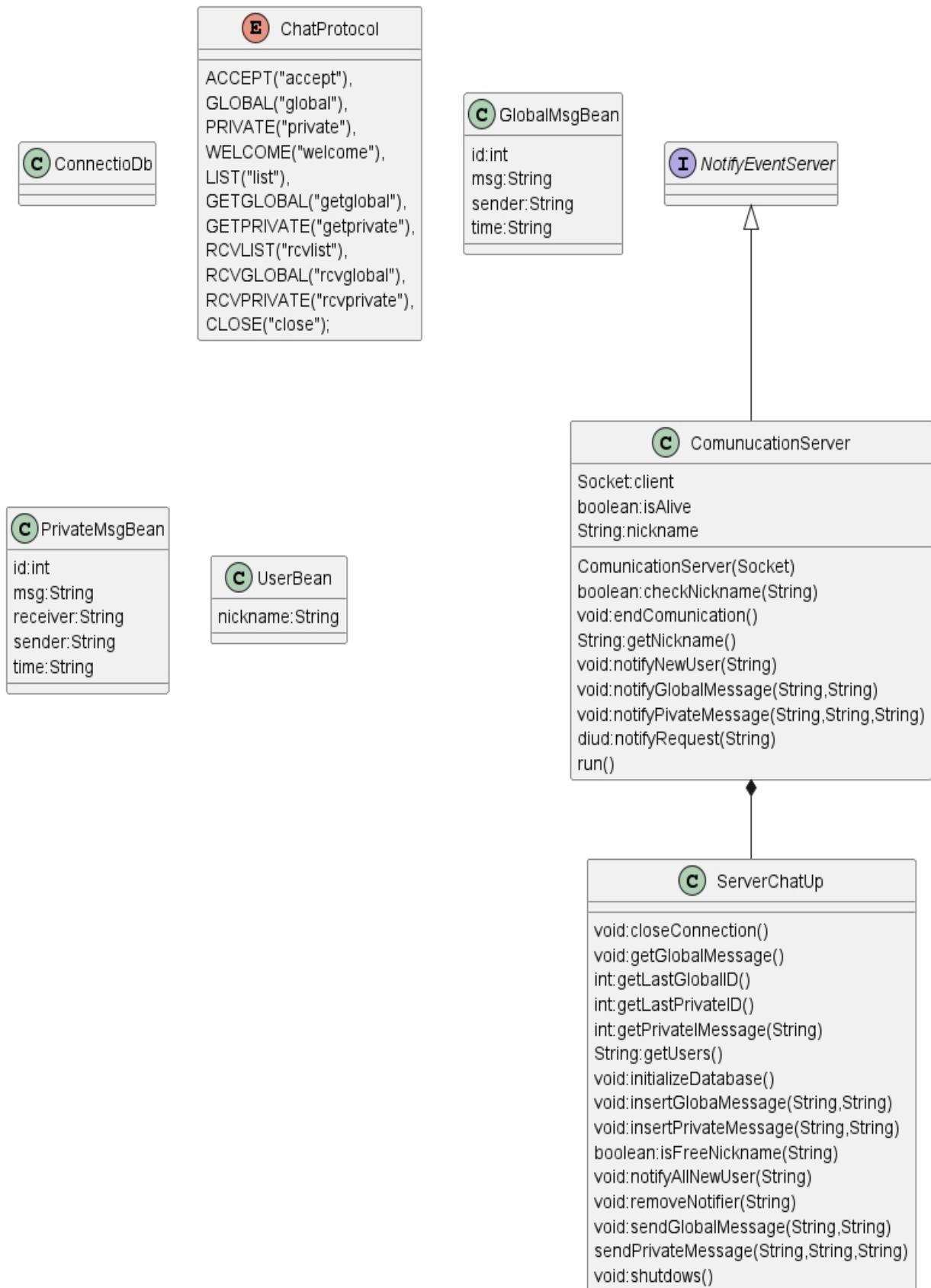
In quest'altro esempio possiamo scorrere tutti i valori dell'enumerazioni.

Puoi utilizzare un'enumerazione in un programma Java per migliorare la leggibilità del codice e rendere più chiaro l'uso di valori costanti che appartengono a un insieme specifico. Le enumerazioni possono anche contenere metodi e campi aggiuntivi, rendendole più flessibili rispetto alle costanti tradizionali.

Le enumerazioni sono spesso utilizzate in situazioni in cui c'è un insieme fisso e noto di valori, come i giorni della settimana, i mesi, le opzioni di menu, ecc.

DIAGRAMMA DELLE CLASSI

Vediamo ora come possiamo progettare le classi del nostro applicativo, utilizzeremo un altro diagramma UML il class diagram



*****ebook converter DEMO Watermarks*****

Il cuore della nostra applicazione è la classe ServerChatup, che conterrà il metodo main.

Questa classe aprirà un SocketServer su una determinata porta nel nostro caso 1612, ogni client che si connetterà a questa porta innescherà l'istanza di una nuova classe ComunicationServer, questa classe contiene il Socket verso il client, ed è delegato a comunicare con esso, inoltre a questa classe è demandato il compito di riconoscere i comandi del protocollo che abbiamo scelto, e rispondere nel modo corretto.

Sviluppo del codice

ecco il codice della nostra classe ServerChatUp

```
package sample.chatup.server;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import sample.chatup.model.GlobalMsgBean;
import sample.chatup.model.PrivateMsgBean;
import sample.chatup.model.UserBean;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class ServerChatUp {
    private static final List<NotifyEventServer> listConnection;
    private static boolean isAlive = false;

    private static Logger logger = LoggerFactory.getLogger(ServerChatUp.class);

    static {
        isAlive = true;
        listConnection = new ArrayList<>();
    }

    public static void main(String[] args) {
        try {
            Connection connection = ConnectioDb.getConnectionInstance();
            Statement stm = connection.createStatement();

            DatabaseMetaData metaData = connection.getMetaData();
            ResultSet tables = metaData.getTables(null, null, null, new String[] { "TABLE" });
        }
    }
}
```

```

        if (!tables.next()) {
            logger.info("Nessuna tabella è stata creata nel database.");
            initializeDatabase();
        } else {
            logger.info("Sono state create tabelle nel database.");
        }

        stm.close();

        ServerSocket server = new ServerSocket(1612);
        while(isAlive) {
            Socket client = server.accept();
            listConnection.add(new CommunicationServer(client));
        }
    } catch (Exception e) {
        logger.error("ERROR",e);
    }
    finally {
        try {
            closeConnection();
        } catch (DBException e) {
            logger.error("Error:",e);
        }
    }
}

private static void initializeDatabase() throws DBException {
    Connection connection = ConnectioDb.getConnectionInstance();
    Statement stm = null;
    try {
        stm = connection.createStatement();
        stm.executeUpdate("CREATE TABLE GLOBAL_MSG(id integer,time timestamp,msg varchar(512),sender varchar(256));");
        stm.executeUpdate("CREATE TABLE PRIVATE_MSG(id integer,time timestamp,msg varchar(512),sender varchar(256),receiver varchar(256));");
    } catch (SQLException e) {
        logger.error("Error:",e);
        throw new DBException(e);
    }
}

```

```

private static void closeConnection() throws DBException{
    try {
        ConnectioDb.getConnectionInstance().close();
    } catch (SQLException e) {
        logger.error("Error:",e);
        throw new DBException(e);
    }
}

public static synchronized boolean isFreeNickname(String nickname){
    for (NotifyEventServer notifier: listConnection){
        if(notifier.checkNickname(nickname)){
            return false;
        }
    }

    return true;
}

public static synchronized void removeNotifier(NotifyEventServer notifier) {
    listConnection.remove(notifier);
}

public static synchronized void notifyAllNewUser(String nickname) throws IOException {
    for (NotifyEventServer notifier: listConnection){
        notifier.notifyNewUser(nickname);
    }
}

public static synchronized void sendGlobalMessage(String message, String nickname) throws
DBException {
    for (NotifyEventServer notifier: listConnection){
        try {
            insertGlobaMessage(message,nickname);
            notifier.notifyGlobalMessage(message,nickname);
        } catch (IOException e) {
            throw new DBException(e);
        }
    }
}

```

```

    public static synchronized void sendPrivateMessage(String message, String nickname, String to)
throws DBException {
    for (NotifyEventServer notifier: listConnection){
        if(notifier.checkNickname(to)){
            try {
                insertPrivateMessage(message, nickname,to);
                notifier.notifyPrivateMessage(message,nickname,to);
            } catch (IOException e) {
                throw new DBException(e);
            }
        }
    }
}

public static void shutdowns(){
    isAlive = false;
}

public static synchronized String getUsers() {
    StringBuilder users = new StringBuilder();
    for (NotifyEventServer notifier: listConnection){
        users.append(new UserBean(notifier.getNickname()));
    }
    return users.toString();
}

public static synchronized String getGlobalMessage() throws DBException {
    StringBuilder messages = new StringBuilder();
    Connection conn = ConnectioDb.getConnectionInstance();

    try {
        Statement stm = conn.createStatement();
        ResultSet rs = stm.executeQuery("SELECT * FROM GLOBAL_MESSAGE");

        while(rs.next()){
            int id = rs.getInt("id");
            String msg = rs.getString("msg");
            Timestamp time = rs.getTimestamp("time");
            String sender = rs.getString("sender");
            GlobalMsgBean bean = new GlobalMsgBean(id,msg,sender,time.toString());
            messages.append(bean.toString());
        }
    } catch (SQLException e) {
        throw new DBException(e);
    }
}

```

```

        }

        return messages.toString();
    }

    public static synchronized String getPrivateMessage(String nickname) throws DBException {
        StringBuilder messages = new StringBuilder();
        Connection conn = ConnectioDb.getConnectionInstance();

        try {
            PreparedStatement stm = conn.prepareStatement("SELECT * FROM PRIVATE_MESSAGE
WHERE sender=? OR receiver=?");
            stm.setString(1,nickname);
            stm.setString(2,nickname);

            ResultSet rs = stm.executeQuery();

            while(rs.next()){
                int id = rs.getInt("id");
                String msg = rs.getString("msg");
                Timestamp time = rs.getTimestamp("time");
                String sender = rs.getString("sender");
                String receiver = rs.getString("receiver");

                PrivateMsgBean bean = new PrivateMsgBean(id,msg, sender, receiver
, time.toString());
                messages.append(bean.toString());
            }
        } catch (SQLException e) {
            throw new DBException(e);
        }

        return messages.toString();
    }

    private static void insertPrivateMessage(String message, String nickname, String to) throws
DBException {
        Connection conn = ConnectioDb.getConnectionInstance();

        try {
            PreparedStatement stm = conn.prepareStatement("INSERT INTO
PRIVATE_MESSAGE(id,time,msg, sender, receiver) VALUES(?,?,?,?,?)");
            stm.setInt(1,getLastPrivateID()+1);
            stm.setTimestamp(2,new Timestamp(System.currentTimeMillis()));
            stm.setString(3,message);
            stm.setString(4,nickname);
            stm.setString(5,to);
        }
    }
}

```

```

        stm.executeUpdate();
    } catch (SQLException e) {
        throw new DBException(e);
    }
}

private static void insertGlobaMessage(String message, String nickname) throws DBException {
    Connection conn = ConnectioDb.getConnectionInstance();

    try {
        PreparedStatement stm = conn.prepareStatement("INSERT INTO
GLOBAL_MESSAGE(id,time,msg,sender) VALUES(?, ?, ?, ?)");
        stm.setInt(1,getLastGlobalID()+1);
        stm.setTimestamp(2,new Timestamp(System.currentTimeMillis()));
        stm.setString(3,message);
        stm.setString(4,nickname);

        stm.executeUpdate();
    } catch (SQLException e) {
        throw new DBException(e);
    }
}

private static int getLastPrivateID() throws DBException {
    Connection conn = ConnectioDb.getConnectionInstance();
    int id = 0;
    try {
        Statement stm = conn.createStatement();
        ResultSet rs = stm.executeQuery("SELECT MAX(id) FROM PRIVATE_MESSAGE");

        while (rs.next()){
            id = rs.getInt(1);
        }
    } catch (SQLException e) {
        throw new DBException(e);
    }
    return id;
}

private static int getLastGlobalID() throws DBException {
    Connection conn = ConnectioDb.getConnectionInstance();
    int id = 0;
    try {
        Statement stm = conn.createStatement();
        ResultSet rs = stm.executeQuery("SELECT MAX(id) FROM GLOBAL_MESSAGE");

```

```

        while (rs.next()){
            id = rs.getInt(1);
        }

    } catch (SQLException e) {
        throw new DBException(e);
    }
    return id;
}
}

```

Vediamo una panoramica delle funzionalità principali:

- **Inizializzazione del Server:**

Viene creato un oggetto ServerSocket che accetta le connessioni sulla porta 1612.

Viene stabilita una connessione al database chiamando ConnectioDb.getConnectionInstance().

- **Inizializzazione del Database:**

Viene verificato se sono presenti tabelle nel database. Se non ci sono tabelle, il metodo initializeDatabase () crea due tabelle nel database: GLOBAL_MSG e PRIVATE_MSG.

- **Accettazione delle Connessioni Client:**

Il server entra in un ciclo while (while(isAlive)) in cui accetta connessioni dai client tramite server.accept(). Per ogni connessione accettata, viene creato un oggetto CommunicationServer e aggiunto a listConnection.

- **Gestione degli Utenti:**

Il server tiene traccia degli utenti connessi tramite la lista listConnection.

Il metodo isFreeNickname(String nickname) controlla se un determinato nickname è disponibile.

Il metodo getUsers() restituisce la lista degli utenti connessi.

- **Invio di Messaggi:**

Il server gestisce l'invio di messaggi globali e privati.

I messaggi globali vengono inseriti nella tabella GLOBAL_MSG e notificati a tutti gli utenti connessi.

I messaggi privati vengono inseriti nella tabella PRIVATE_MSG e inviati solo al destinatario specificato.

- **Recupero dei Messaggi:**

Il server fornisce metodi (getGlobalMessage() e getPrivateMessage(String nickname)) per recuperare i messaggi globali e privati dal database.

- **Chiusura del Server:**

Il server può essere chiuso chiamando il metodo shutdown(), il che imposta isAlive su false.

- **Gestione delle Eccezioni:**

Le eccezioni vengono catturate e registrate tramite l'oggetto Logger.

- **Metodi di Supporto:**

Ci sono vari metodi di supporto per l'inserimento di messaggi nel database, il recupero dell'ultimo ID e la chiusura della connessione al database.

- **Utilizzo del Modello:**

Il codice utilizza modelli come **GlobalMsgBean**, **PrivateMsgBean**, e **UserBean** per rappresentare rispettivamente i messaggi globali, i messaggi privati e gli utenti. Questi modelli contengono informazioni come ID, messaggio, mittente, destinatario e timestamp.

Questa invece è l'implementazione della classe CommunicationServer

```
package sample.chatup.server;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.net.Socket;

public class ComunicationServer implements Runnable,NotifyEventServer{

    private final Socket client;
    private boolean isAlive;
    private static final String PROTOCOL_SEP = "@";
    private static final String PROTOCOL_VERSION = "1.0";

    private String nickname ;
    public ComunicationServer(Socket client){
        this.client = client;
        isAlive = true;
    }

    /**
     *
     */
    @Override
    public void run() {
        try {
            DataOutputStream output = new DataOutputStream(client.getOutputStream());
            PrintStream ps = new PrintStream(output);
            DataInputStream input = new DataInputStream(client.getInputStream());
            byte[] message = new byte[1000];
            int nb = input.read(message);
        }
    }
}
```

```

        if(nb== -1){
            ps.println("Connection refused");
            ServerChatUp.removeNotifier(this);
            client.close();
            return;
        }

        String accept = new String(message);
        String[] command = accept.split("\\"+PROTOCOL_SEP);

        if(command.length!=2){
            ps.println("Connection refused protocol unknown");
            ServerChatUp.removeNotifier(this);
            client.close();
            return;
        }
        if(ChatProtocol.ACCEPT.getProtocol().equals(command[0])){
            if(!ServerChatUp.isFreeNickname(command[1])){
                ps.println("Connection refused nickname already exist");
                ServerChatUp.removeNotifier(this);
                client.close();
                return;
            }
            else {
                this.nickname=command[1];
                ServerChatUp.notifyAllNewUser(nickname);
            }
        }
        else {
            ps.println("Connection refused protocol unknown");
            ServerChatUp.removeNotifier(this);
            client.close();
            return;
        }

        ps.println(ChatProtocol.WELCOME +PROTOCOL_SEP+PROTOCOL_VERSION);
        while(isAlive){
            int n = input.read(message);
            if(n== -1){
                continue;
            }
            String msg = new String(message);
            String[] protoMsg = msg.split("\\"+PROTOCOL_SEP);

            if(ChatProtocol.GLOBAL.getProtocol().equals(protoMsg[0])){
                ServerChatUp.sendGlobalMessage(protoMsg[1],nickname);
            }
            else if (ChatProtocol.PRIVATE.getProtocol().equals(protoMsg[0])){
                ServerChatUp.sendPrivateMessage(protoMsg[1],nickname,protoMsg[0]);
            }
            else if(ChatProtocol.LIST.getProtocol().equals(protoMsg[0])){
                String users = ServerChatUp.getUsers();

```

```

        notifyRequest(ChatProtocol.RCVLIST.getProtocol() + PROTOCOL_SEP + users);
    }
    else if(ChatProtocol.GETGLOBAL.getProtocol().equals(protoMsg[0])){
        String global = ServerChatUp.getGlobalMessage();
        notifyRequest(ChatProtocol.RCVGLOBAL.getProtocol() + PROTOCOL_SEP + global);
    }
    else if(ChatProtocol.GETPRIVATE.getProtocol().equals(protoMsg[0])){
        String privMsg = ServerChatUp.getPrivateMessage(nickname);
        notifyRequest(ChatProtocol.RCVPRIVATE.getProtocol() + PROTOCOL_SEP + privMsg);
    }
    else if(ChatProtocol.CLOSE.getProtocol().equals(protoMsg[0])){
        ServerChatUp.removeNotifier(this);
        client.close();
        return;
    }
}

} catch (IOException e) {
    throw new RuntimeException(e);
} catch (DBException e) {
    throw new RuntimeException(e);
}

}

/***
 * @param user
 */
@Override
public void notifyNewUser(String user) throws IOException {
    DataOutputStream output = new DataOutputStream(client.getOutputStream());
    PrintStream ps = new PrintStream(output);
    ps.println(ChatProtocol.ACCEPT + PROTOCOL_SEP + user);
}

/***
 * @param message
 */
@Override
public void notifyGlobalMessage(String message, String from) throws IOException {
    DataOutputStream output = new DataOutputStream(client.getOutputStream());
    PrintStream ps = new PrintStream(output);
    ps.println(ChatProtocol.GLOBAL + PROTOCOL_SEP + message + PROTOCOL_SEP + from);
}

/***
 * @param message
 */

```

```

@Override
public void notifyPrivateMessage(String message, String from, String to) throws IOException {
    DataOutputStream output = new DataOutputStream(client.getOutputStream());
    PrintStream ps = new PrintStream(output);
    ps.println(ChatProtocol.PRIVATE+PROTOCOL_SEP+message+PROTOCOL_SEP+to);
}

public void notifyRequest(String message) throws IOException {
    DataOutputStream output = new DataOutputStream(client.getOutputStream());
    PrintStream ps = new PrintStream(output);
}

/**
 * @param message
 */
/**
 */
/**
 *
 */
/**
 */

@Override
public void endCommunication() {
    isAlive = false;
}

/**
 * @param nickname
 * @return
 */
@Override
public boolean checkNickname(String nickname) {
    return this.nickname.equals(nickname);
}

@Override
public String getNickname() {
    return nickname;
}

}

```

La classe ComunicationServer è responsabile della gestione della comunicazione tra il server principale (ServerChatUp) e un singolo client. Ecco una spiegazione delle principali funzionalità della classe:

- **Costruttore:**

Il costruttore accetta un oggetto Socket che rappresenta la connessione con un client. Inizializza anche il flag isAlive a true.

- **Metodo run:**

Questa classe implementa l'interfaccia Runnable, il che significa che il suo comportamento principale è definito nel metodo run().

Inizializza gli stream di input e output per comunicare con il client.

Legge il primo messaggio inviato dal client e verifica il protocollo di connessione.

Se il protocollo è accettato, verifica la disponibilità del nickname fornito dal client. Se il nickname è disponibile, notifica agli altri utenti che un nuovo utente è entrato nella chat.

Se la connessione è riuscita, il server invia un messaggio di benvenuto al client.

Successivamente, entra in un loop in cui legge continuamente i messaggi inviati dal client e agisce di conseguenza, ad esempio inviando messaggi globali, privati, richiedendo la lista degli utenti o chiudendo la connessione

- **Metodi di Notifica:**

La classe implementa l'interfaccia **NotifyEventServer** che richiede l'implementazione di alcuni metodi di notifica. In particolare:

notifyNewUser: Notifica al client che un nuovo utente è entrato nella chat.

notifyGlobalMessage: Notifica al client l'arrivo di un nuovo messaggio globale.

notifyPrivateMessage: Notifica al client l'arrivo di un nuovo messaggio privato.

notifyRequest: Metodo vuoto che sembra essere incompleto.

- **Metodo endCommunication:**

Imposta il flag isAlive a false, indicando che la comunicazione deve terminare.

- **Metodo checkNickname:**

Verifica se il nickname fornito come argomento è uguale al nickname associato a questa istanza di ComunicationServer.

- **Metodo getNickname:**

Restituisce il nickname associato a questa istanza di ComunicationServer.
la classe gestisce la comunicazione tra il server principale e un singolo client, interpretando i messaggi inviati dal client e rispondendo di conseguenza. È progettata per essere eseguita in un thread separato per gestire più connessioni contemporaneamente.

LE ANNOTATION

Nel nostro progetto abbiamo utilizzato delle stringhe fisse per comunicare un record, questo sistema permette di non utilizzare caratteri speciali per dividere i campi, un utile esercizio potrebbe essere quello di creare un meccanismo che definisca un convertitore da stringa fissa a bean, per poter effettuare l'elaborazione avremo bisogno dei campi che potrebbero essere recuperati tramite il meccanismo di reflection.

La reflection in Java è una caratteristica che consente al programma di ispezionare o manipolare il comportamento di classi, metodi, campi e altri elementi di un programma durante l'esecuzione. In sostanza, consente di esaminare e interagire con la struttura del programma durante il runtime anziché durante la compilazione.

Le principali classi coinvolte nella reflection sono contenute nel pacchetto `java.lang.reflect`.

Oltre i campi avremo bisogno di definire una lunghezza per ognuno di essi, ma come è possibile modellare il nostro bean inserendo anche le informazioni della lunghezza per ogni campo?

Per questo ci viene in aiuto le annotation.

In Java, le annotazioni (annotations) sono metadati che possono essere aggiunti a codice, dichiarazioni di classe, metodi, campi, parametri e altri elementi del codice sorgente per fornire informazioni aggiuntive sull'elemento annotato. Le annotazioni in Java iniziano con il simbolo `@` seguito dal nome dell'annotazione.

Le annotazioni sono utilizzate principalmente per fornire informazioni di configurazione, istruzioni per il compilatore, e per l'integrazione con strumenti di analisi del codice o framework. Le annotazioni non influiscono direttamente sul comportamento del programma, ma possono essere lette durante l'esecuzione o elaborazione del codice.

Ecco alcune delle annotazioni integrate nel linguaggio Java:

`@Override`:

Indica che un metodo deve sovrascrivere un metodo nella sua classe genitore. Utile per prevenire errori di firma di metodo.

```
@Deprecated  
public class OldClass {  
    // implementazione della classe  
}
```

@SuppressWarnings:

Ignora gli avvertimenti del compilatore per un particolare elemento o per l'intera classe.

```
@SuppressWarnings("unchecked")  
public List<String> getList() {  
    // implementazione del metodo  
}
```

@FunctionalInterface:

Indica che un'interfaccia ha esattamente un metodo astratto e può essere trattata come un'interfaccia funzionale.

```
@FunctionalInterface  
public interface MyFunctionalInterface {  
    void myMethod();  
}
```

@Entity, @Table, etc. (JPA Annotations):

Usate in Java Persistence API (JPA) per mappare classi Java a tavole del database.

```
@Entity  
@Table(name = "my_table")  
public class MyClass {  
    // implementazione della classe  
}
```

Definizione di Annotazioni Personalizzate

È possibile creare annotazioni personalizzate definendo nuove annotazioni con la parola chiave @interface.

```
@interface MyAnnotation {  
    String value();  
    int count() default 1;  
}  
  
@MyAnnotation(value = "example", count = 3)  
public class MyClass {  
    // implementazione della classe  
}
```

Adesso che abbiamo capito cosa sono le annotation, vediamo come applicarle al nostro caso, realizzeremo dei bean che hanno una annotation sulla definizione dei campi, che ci diranno anche quando il campo deve essere lungo:

questa è la definizione della nostra annotation che chiameremo FixedField

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface FixedField {
    17 usages
    public int size();
}
```

L'annotation `@Retention` ci permette di specificare che la nostra custom annotation lavorerà in fase di runtime, mentre target specificherà che la dichiarazione è sul campo, altri varianti sono method e type(sulla dichiarazione della classe).

Ora vediamo come applicare la nostra annotation su un bean:

```
@FixedField(size = 256)
private String nickname;
```

In questo caso il campo nickname viene annotato con `@FixedField` che specificherà una lunghezza di 265 caratteri.

RIFERIMENTI

Per qualsiasi chiarimento è possibile contattare l'autore all'indirizzo email
ppassantini@gmail.com, i sorgenti del progetto Chatup si trovano
<https://github.com/pietropassantini/chatup>