APPUNTI POO

4 novembre prima prova intercorso

Durata corso 12 settimane

Libro: Concetti di informatica e fondamenti di java

Ambiente di sviluppo JDK 15 + documentazione

IDE da scaricare: Eclipse + Maven

POO si basa su PSD con astrazione, information hiding, il riuso del codice e la modularità, promosse con tecniche diverse di programmazione.

L’astrazione è un processo mentale che porta ad evidenziare i punti cardine del problema, ignorando aspetti ininfluenti e secondari

L’ astrazione poi può essere funzionale: ovvero riconoscendo aspetti che possono essere affidati a sottoprogrammi che potranno anche essere riutilizzati in futuro(riuso del software e modularità) e poi l’astrazione sui dati

In C++ è possibile definire funzioni della definizione di struct, così che solo quella struttura possa usare quelle funzioni, a differenza di c che vieta questa modularità

In Java una classe è una struttura dati in cui i dati sono accessibili solo dalle funzioni definite nella struttura

La programmazione ad oggetti si basa su un paradigma in cui si possono definire oggetti software, istanze di classi in grado di interagire le une con le altre attraverso lo scambio di messaggi

Una classe è un template per creare un oggetto, l’oggetto è la concretizzazione del modello descritto nella classe

La classe presenta un nome, degli attributi (caratteristiche della classe e variabili usate) e dei metodi (operazioni possibili)

Java a differenza di C++ è un linguaggio puramente ad oggetti anche se la loro sintassi è simile in più i programmi in java si astraggono dal sistema operativo a differenza di C++

C++ non è portabile, in quanto il compilato in windows funziona solo lì e non su altre macchine, mentre Java è più “universale”

In più in C++ se il programma compilato fallisce, si ha soltato un errore e non viene specificato dove o per cosa

**Java è un linguaggio INTERPRETATO**, ovvero il programma durante l’interpretazione e l’esecuzione è in grado di riconoscere l’errore e la causa, in più a differenza di C++ è portabile, il contro di Java è la sua inefficienza

La portabilità di Java deriva dalla creazione e l’esecuzione del programma in una JVM (Java Virtual Machine) quindi si astrae dal sistema operativo usato dal programmatore e dall’utente

Java sources (codice sorgente) -> JAVAC -> Byte Code (codice comprensibile all’interprete) -> Java

I programmi in Java sono composti da entità come variabili, routine e istruzioni:

* Una variabile cratterizzata da nome, tipo e un’area di memoria per la memorizzazione del valore
* Una routine è caratterizzata da un nome, da parametri di un tipo specifico e
* Un’istruzione è caratterizzata da azioni associate

Il binding è l’associazione del tipo della variabile alla variabile stessa, i tipo di binding possono essere:

- l’associazione del tipo di linguaggio da usare

- Binding durante l’implementazione del linguaggio -> l’associazione di un tipo ad una variabile per riservare una locazione di memoria ad un tipo specifico di rappresentazione

- binding durante l’esecuzione di un programma -> associazione di un tipo di dato ad una variabile durante l’esecuzione di un programma, in quanto a priori al programma è stato definito che potrebbe contenere certi tipi di dati, sta poi all’interprete capire quale scegliere e che tipo di rappresentazione usare in quella locuzione di memoria

Il binding può essere dinamico(stabilito durante l’esecuzione del programma) o statico(stabilito prima dell’esecuzione del programma)

Una variabile in java si definisce con:

**nome scope tipo l-value r-value**

dove:

scope = visibilità

l-value = indirizzo in memoria

r-value = valore della variabile memorizzato nella locazione indicata dall’L-value

Python ha una tipizzazione debole, un tipo definito come stringa può diventare integer in base al suo valore.

Es. str= “ciao” è una stringa ma str=5 è un integer

L’esecuzione di ogni programma in java non avviene su una macchina ma su un’astrazione di questa, rendendolo portatile

Le api sono librerie, ordinate con classi

L’ordine interno delle librerie è organizzato in package

JRE = java runtime enviroment -> serve ad eseguire i programmi

SDK = software development kit -> per programmare

Tipo enumerativo (Enum): **ESAME**

Viene utilizzata molto spesso per diminuire il codominio, ovvero l’insieme dei valori che possono essere memorizzati in una variabile, a differenza di c, in java Enum è un oggetto che viene utilizzato come tipo, in quanto non esistono tipi fondamentali enumerativi in java, in più non viene memorizzato attraverso tipi di supporto ma utilizza oggetti (immutabili) già creati, in Java quindi Enum è una classe (java.lang.Enum), che come vantaggio ha delle funzioni pronte già nel package

Casting: tipo a -> tipo b

Parsing: “b” -> b

Gli errori in java sono eccezioni e se è missmatchato il case di un oggetto si ottiene java.lang.IllegalArgumentException”

Enum essendo una classe può contenere metodi

In un enumerativo i campi devono essere privati e non cambiano nei vari oggetti, ma da istanza ad istanza

throw è una funzione che solleva di priorità una funzione, come un interrupt software, nel caso di throw new AssertionError l’“interrupt” sale di priorità sopra il main al punto di bloccare l’esecuzione del programma per eseguire l’avviso scritto, questo è utile per gestire le eccezioni

**Array**

Accesso posizionale attraverso []

A differenza di C, l’allocazione è dinamica attraverso l’operatore **new**

In java è possibile anche creare array multidimensionali

Per operare sugli array si utilizza la classe java.lang.Arrays

**Garbage Collector**

Nella JVM a differenza di C, non esistono malloc e free, ma si istanzia un oggetto e viene poi “distrutto” dal garbage collector, ovvero viene raccolto dall’heap un oggetto che non viene utilizzato, è utile per poter evitare errori come segmentation fault in c++ o c

Il finalize è un metodo per *rilasciare risorse*, che viene eseguito alla chiusura di uno stream/risorsa (file, connessione…) e viene chiamato dal garbage collector prima di deallocare l’oggetto. Viene quindi prima rilasciata la risorsa dell’oggetto e poi deallocato l’oggetto

Nell’heap è memorizzato un intero detto reference counter che conta quanti nello stack puntano ad un oggetto, il garbage collector, scorrendo l’heap e vedendo un reference counter dealloca l’oggetto non più utilizzato. Questi oggetti vengono prima marcati e poi parte la deallocazione.

Vengono poi distinti gli oggetti in base alla loro creazione, dividendoli in 3 generazioni:uQ

* Generazione nuova =
* oggetti creati da poco
* Generazione vecchia = oggetti creati da un po’ che permangono nell’esecuzione del programma
* Generazione permanente

Ogni generazione dipende da quanti cicli di *pulizia* è sopravvissuto

L’overloading sulle funzioni serve per poter operare con funzioni concettualmente simili ma applicabili su tipi di dato diversi, es. una funzione f1 sui char e una f2 sugli interi, viene fatto l’overloading con la funzione adatta al dato corrente

Programmazione generica = programmazione astratta sui dati per poter permettere di eseguire un algoritmo su tipi diversi di dati

Il fine quindi della programmazione generica è di scrivere un algoritmo una sola volta, piuttosto che scriverne una variante per ogni tipo di dato, che sia l’applicazione dell’algoritmo o dei dati

Prima di Java 5, la tecnica utilizzata era quella di utilizzare un tipo Object che andava castato esplicitamente al tipo di dato che si volevano usare. Da Java 5 è stato introdotto Java Generics, che rende possibile il controllo sintattico del generico tipo utilizzato, rendendo possibile un controllo sintattico-semantici, gli errori quindi possono essere risolti a tempo di compilazione piuttosto che a tempo di esecuzione come con gli Object, in Generics non è più necessario il casting esplicito in quanto esistono e possono essere specificati degli .

Es. List <String> list = new ArrayList<String>()

Dove string indica il tipo che deve essere considerato quando poi avvengono interazioni con altre variabili, rendendo inutile il casting esplicito

Poi per essere utilizzati, vengono specificati i tipi parametrizzati quando vengono utilizzati

Tipo avendo una classe con un tipo generico:

es. Box<Integer> IntegerBox = new Box<Integer>(); --può anche essere sostituito da--> Box<Integer> IntegerBox = new Box<>();

Tutti i tipi generici per via dell’uso di Object sono definiti tipo RAW, avendo gli stessi problemi della mancanza di controlli sintattico-semantici degli Object prima di Generics, per via di questa mancanza di controlli può portare ad errori a tempo di esecuzione se si provano ad assegnare dei valori Raw ad altri tipi

La genericità è un polimorfismo statico, mentre l’ereditarietà è un polimorfismo dinamico

**Wildcard**

A tempo di scrittura del codice, si può utilizzare l’operatore “?” per indicare un tipo di dato sconosciuto, utilizzato per rendere dinamico un polimorfismo statico, non si usa come argomento per la creazione di una classe perché non è reificato e non si sa quanto spazio in memoria debba occupare per memorizzare informazioni, perciò non possono essere istanziati (non si può usare new classe ? ), però tipi non reificati possono essere utilizzati come reference (come il void\* in c)

Anche le Wildcard possono essere limitate superiormente con extends:

list<?extends Number> indica tutti i sottoinsiemi di Number

list<Number> indica soltanto Number senza Integer, Float o Double

Il tipo wildcard (?) viene assegnato a tempo di esecuzione e non di compilazione

La cattura della wildcard serve per poter avere *qualcosa* di compilabile ma comunque generico a tempo di esecuzione così da aiutare l’inferenza dalla wildcard a CAP#1

Per le variabili di ingresso si limitano inferiormente le wildcard

Per le variabili di uscita si limitano superiormente

Se la variabile di ingresso possono usare metodi della classe object

La TypeErasure (cancellazione del tipo) durante la compilazione sostituisce tutti i tipi con Object e i vincoli specificati o semplicemente in Object se non vincolati

Un tipo si dice reificabile se a tempo di esecuzione sono disponibili le sue informazioni, questo tipo comprende:

* I primitivi
* I non generici
* Raw
* Wildcards non vincolate

**Reflection**

È la capacità di un programma di vedere sé stesso, in modo di poter manipolare strutture, metodi, oggetti e classi a tempo di esecuzione, sotto java.lang.reflection sono istanze di classi per l’interrogazione dinamica.

Es.

**Senza reflection:**

Foo foo = new Foo();

foo.hello();

**Con refletion**:

Object foo = Class.forName(“complete.classhpath.and.Foo”).newInstance();

**Oppure:**

Object foo = Foo.class.newInstance();

Method m = foo.getClass().getDeclaredMethod(“hello”,null);

m.invoke(foo);

Esistono reflection:

* Strutturale = sulle parti che lo compongono

Letterale di classe = “nome di una classe . e dalla parola class” es. Integer.class

Le annotazioni (@) danno informazioni al compilatore in modo che capisca cosa fare prima della compilazione, per esempio @Override da l’informazione al compilatore di controllare che esista un metodo o una classe con la sintassi corrispondente a quella che stiamo scrivendo per overridare

A loro volta le annotazioni possono avere dei campi come per esempio

@SuppressWarnings( “unchecked” ) void myMethod() {…}

Eager è un’allocazione che avviene alla dichiarazione dell’espressione, mentre lazy l’allocazione in cui un’espressione viene valutata solo quando è richiesto il suo valore e non quando “appare per la prima volta”

Optional è un tipo in java che può supportare il tipo nullo per evitare il null.pointer.exeption vengono usati nel caso il valore di ritorno di una funzione può essere nullo, quindi Optional può ospitare un valore effettivo o un valore null.

Optional è utile per evitare di tanti controlli sui valori null, in quanto vengono gestiti dal programma direttamente senza l’uso di if

.filter utilizza un’iterazione sugli elementi di uno stream per, appunto, filtrare gli elementi, per poi computare con ognuno il predicato. Nel caso poi si voglia trovare un solo elemento dello stream, .filter una volta trovato quell’elemento continua ad iterare, causando uno spreco di memoria; in alternativa si può usare takeWhile che computa il predicato ed itera sugli elementi fino a trovare un elemento che non soddisfa il predicato, questo causa un tempo di latenza minore.

Inversamente da takeWhile, skipWhile salta gli elementi finché non ne trova uno che non soddisfa il predicato

dropWhile cancella gli elementi finché non è soddisfatto il predicato

su uno stream è possibile effettuare una ricerca, attraverso **anyMatch** che ritorna vero se almeno un elemento dello stream soddisfa il predicato

**allMatch** se **tutti** gli elementi soddisfano il predicato ritorna vero, falso altrimenti

**noneMatch** se nessuno degli elementi soddisfa il predicato ritorna vero

**findFirst** ritorna il primo elemento dello stream

**findAny** ritorna un elemento che soddisfa il predicato

nella classe optional<T> ci sono:

**ifPresent** che ritorna vero se è presente un valore nello stream

**T get** restituisce il valore se presente, altrimenti un’eccezione NoSuchElementException

**T orElse(T other)** restituisce il valore T se presente, altrimenti restituisce “T other”

Il parallelismo può causare un rallentamento a carichi piccoli, perciò la JVM con parallelstream se si ha un carico piccolo, non spreca tempo e memoria per creare, gestire e “disposare” i thread

**SECONDA PARTE POO**

Polimorfismo

Vengono create delle astrazioni (oggetti) per poter riutilizzare algoritmi o strutture, questo è il fine della programmazione ad oggetti.

Protected = scope limitata alla classe e alle sue sottoclassi  
Private = scope limitata solo alla classe che dichiara la variabile in questa modalità

Le superclassi che non hanno bisogno di essere utilizzate possono essere dichiarate abstract costringendo l’utente ad utilizzare una sottoclasse che le estende

Super all’interno di una sottoclasse funziona da riferimento alla superclasse, è utile per poter invocare metodi della classe padre nei metodi o in generale nella classe figlio, tipo se abbiamo una classe “padre” e una classe “figlio”, figlia della prima:

class figlio extends padre {

super.*metodo()* } corrisponde a -> padre.*metodo()*

Le variabili locali e i dati locali vengono memorizzate nello stack, mentre i dati della superclasse sono memorizzati nell’heap, che è accessibile grazie al puntatore della sottoclasse (con extends)

Ogni oggetto è un dato strutturato, ovvero nella memoria viene allocato lo spazio per tutte le variabili d’istanza

In base al puntatore di ogni oggetto, l’interprete java ricerca a ritroso nella classe alla quale appartiene,

Integer i, per esempio, ha un puntatore (a livello di memoria) alla classe Integer, permettendo così di invocare metodi di Integer con riferimento all’oggetto i attraverso l’operatore punto

**Gestione delle Eccezioni**

Un errore è un comportamento inaspettato, e possono essere:

* Di programmazione: cast non permesso, divisione per zero, accesso oltre i limiti di array
* Di sistema: disco rotto, connessione remota chiusa, memoria non disponibile
* Di utilizzo: Input non corretti, tentativo di lavorare su file inesistente

Throwable è una superclasse di tutti gli errori e le eccezioni e definisce il caso base di ogni errore o eccezione specificata

La classe Error è usata per indicare una situazione anomala nell’esecuzione dell’applicazione fuori dal controllo dell’app stessa, esempio: esaurimento delle risorse della JVM (OutOfMemoryError)

Le eccezioni non controllate non sono obbligatoriamente da gestire, come: **NullPointerException** e **IndexOutOfBoundException**

Le eccezioni possono anche essere progettate, per crearne una non controllata, creiamo una classe che estende RuntimeException

Dove vengono poi specificati dei costruttori, in cui si utilizza il costruttore super con un messaggio di errore tipo *super(“Impossibile dividere per 0”);*

Le eccezioni possono essere lanciate da throw seguita da un oggetto di tipo Exception che terminano immediatamente l’esecuzione del metodo

Gestire un’eccezione = segnalare la sua esistenza o gestire il suo funzionamento con try-catch

Segnalare = throws vicino al nome del metodo seguito poi dall’eccezione che può avvenire in quel metodo

**TRY-CATCH**

Per catturare le eccezioni si usa l’enunciato try sulla parte che può far sorgere eccezioni e tanti catch quante sono le eccezioni a cui dare risposta, es.

Try {

…

Istruzioni

…

}

Catch (TipoEccezione variabile) {

…

Istruzioni

…

}

ecc…

**Flussi/file**

Uno stream non è altro che una sequenza **ordinata** di dati, in cui quelli di input hanno una sorgente e quelli di output una destinazione

Esistono due tipi di flussi:

* Di caratteri (character streams)
* Di dati binari (byte streams)

Ognuno è gestito da una classe apposita

System.in è un flusso di input di tipo InputStream, mentre System.out (come System.err) è un flusso di output di tipo PrintStream (deriva da OutputStream)

Per aprire un flusso di byte da un file si usa la classe **FileInputStream**, e nel costruttore inserire una Stringa che rappresenta il nome del file

Utilizzando il metodo .read() che se il flusso non è terminato (restituisce -1 in quel caso), restituisce un int tra 0 e 255

Per chiudere il flusso si usa il metodo .close(); che rilascia le risorse impegnate per lo stream

Mentre per scrivere si usa la classe **FileOutputStream,** passando nel costruttore il nome del file come stringa

Per scrivere si usa il metodo .write(x) dove x è di tipo byte

Si chiude lo stream con .close();

**APPLICAZIONI GRAFICHE**

È un programma in java che visualizza informazioni all’interno di una finestra ditata di barra di titolo e cornice(frame), ogni frame è gestito da un thread diverso

**Progettazione**

**Analisi:**

definire il fine del programma e cosa deve fare **senza** pensare all’implementazione o al **“come”** farlo

**Progettazione:**

Definisce il **come** implementare il sistema per risolvere il problema e raggiungere il suo fine

**Implementazione:**

Scrivere il codice

Risultato: programma

**Collaudo:**

Verifiche che il programma funzioni correttamente

Risultato: un documento riassuntivo delle operazioni di verifica con i risultati ottenuti

**Installazione:**