PROGRAMMAZIONE 2

LEZIONE 1: EREDITARIETA'

Meccanismi

Linguaggio ad Oggetti ha 3 meccanismi:

- Incapsulamento (e Information hiding),
- Ereditarietà
- Polimorfismo

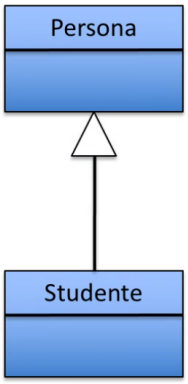
Il problema del primo meccanismo è che si possono avere ripetizioni che

1. Non sono efficienti
2. In caso di errori sono difficili da risolvere, è meglio avere una sola cosa.

Relazione Is-A

Classi Persona e Studente, Studente è un **tipo** di Persona

UML: rappresentata da una freccia bianca



Tutto ciò che rappresenta la classe generica(Persona), sarà anche per la classe più specifica(Studente).

Extends è il costrutto che ci permette di implementare correttamente la relazione is-a. Attraverso extend la classe Studente **eredita** il codice della classe Persona

EREDITARIETÀ

```
public class Persona {
 private String nome;
 private String cognome;

 public Persona(String nome, String cognome) {
    this.nome = nome;
    this.cognome = cognome;
 }
```

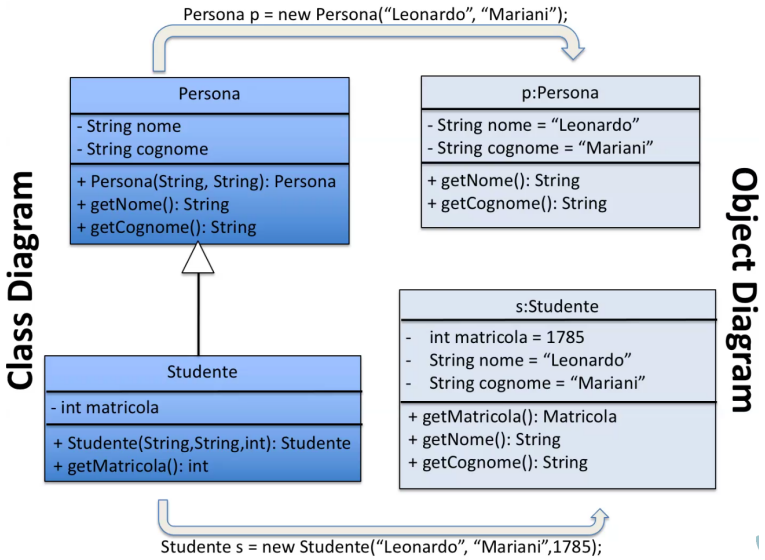
```
public class Studente extends Persona {
    private int matricola;

    public Studente(
     String nome, String cognome, int matricola){
      super(nome, cognome);
      this.matricola = matricola;
    }
```

Ereditarietà è il processo attraverso cui una **classe derivata** viene creata a partire da una **classe base**, rispetto alla relazione generalizzazione-specializzazione.

La classe derivata riusa le variabile/metodi (senza doverle re implementare) della classe base e ne definisce addizionali.

La classe derivata dunque ha le stesse funzionalità della classe Base più altre funzionalità (concetto fondamentale per il Polimorfismo)



N.B: In Java la **ereditarietà multipla non è permessa**, cioè una classe non può estendere da due classi madri. Es. (class Studente lavoratore extends Studente, Lavoratore)

Riassunto ereditarietà

Riuso
- la superclasse può essere riusata in contesti diversi

Economia
- le operazioni comuni vengono scritte una sola volta

OVERRIDING

Sovrascrive il metodo precedente aggiungendo ciò che necessita per il giusto comportamento.

Es: ho la stampa di una Persona con attributi nome e cognome e voglio anche la stampa della matricola se sto lavorando con uno studente.

Notazione **@Override** per specificare che il metodo esegue la ridefinizione di un altro metodo.

la keyword **super** permette di accedere ai metodi della classe base, invece di creare una ricorsione.

```
public String toString(){ return
    "nome=" + nome + " cognome=" + cognome; }
}
```

```
@Override
public String toString(){
  return super.toString() + " matricola=" +
matricola; }
}
```

Overloading	*Overriding*
Diversi metodi con stesso nome ma firme differenti	Ridefinizione del comportamento di un metodo

Final

se il modificatore final precede la definizione di un metodo o di una classe, questo non potrà essere ridefinito nelle classi derivate.

Super, Costruttori

I **costruttori** della classe derivata **devono** invocare un costruttore della classe base.
La keyword per invocare un costruttore della classe base è super(..). Se è omessa si considera una invocazione implicita super() senza parametri

Object

Object è il tipo più generale su Java, contiene alcuni metodi, ad esempio l'equals e toString (ritorna nomeclasse@hashcode). La loro implementazione è troppo generica e vengono tipicamente ridefiniti tramite override.

Quindi ogni volta che utilizzo un metodo equals o toString sto effettuando un override.

Package

Un package è una collezione di classi correlate a cui viene assegnato un nome. Permette di organizzare classi logicamente correlati tra loro e si evitano i conflitti tra nomi delle classi

Attributi dei Metodi

- public: nessuna restrizione
- private: accessibili solo dalla classe di definizione
- protected: non accessibili dall'esterno eccetto le classi derivate e stesso package
- package-wide: accessibili da quelle dello stesso package (default)

LEZIONE 2: POLIMORFISMO

Proprietà del codice di comportarsi in modo diverso in base al contesto dell'esecuzione

- Poliformismo tra reference: reference per riferirsi a oggetti di tipo dinamico diverso da quello dichiarato nel codice (tipo statico).
- Polimorfismo per inclusione: il tipo dinamico di una variabile è limitato dalla corrispondenza tra tipi definita dalle gerarchie di ereditarietà

Es. tra Reference: ho stampa di una Persona che può essere una Persona o un o Studente

Binding dinamico

Binding dinamico/lane binding: legame tra definizione e invocazione di un metodo non viene definito staticamente (compile time) ma dinamicamente (run time).

Nessun binding dinamico per metodi statici sono associati a classi e per metodi final perché non puoi fare overriding.

Casting

Perchè il polimorfismo sia possibile dobbiamo poter assegnare ad una variabile un dato di tipo diverso da quello dichiarato.

UpCasting: coercizione di un tipo specializzato in un tipo più generico (up sta per più alto in gerarchia)

Es. Poligono p=new Rettangolo() , aggiungiProdotto(new CD())

DownCasting: coercizione di un tipo generico in un tipo più specializzato (down sta per più in basso nella gerarchia)

Poligono p = new rettangolo(), rettangolo r, r= (Rettangolo)p

Downcasting è type-unsafe, può dare errore in fase di compilazione.

Operatore instanceof

L'operatore instanceof permette di controllare il tipo dinamico associato ad un reference:
<reference> instanceof <nomeClasse> :

- ritorno true: se il tipo dinamico è compatibile (stesso tipo o sotto tipo)
- ritorno false: altrimenti

metodo getclass()

getClass() è un metodo final della classe Object. Ritorna un tipo dinamico dell'oggetto. Il ritorno può essere confrontato con == e !=.

metodo equals

tutte le classi ereditano equals da Object ma va ridefinito per lo scopo

LEZIONE 3: CLASSE ASTRATTE E INTERFACCIA

CLASSE ASTRATTA

Classe che non rappresenta un concetto concreto: Non esiste un Animale, esistono Cani, Gatti ecc..

Le Classi astratte ***non possono essere istanziate***.
In UML si scrivono con il nome in corsivo.
Keyword su Java: public abstract class nome_classe

Metodo privo di implementazione, non invocabile
Utile per "imporre" l'implementazione di un metodo comune a tutte le sotto-classi di cui non si può dare l'implementazione nella super-classe (Una classe può essere astratta anche se priva di metodi astratti, ma una classe con metodi astratti deve necessariamente essere astratta)

INTERFACCIA

Insieme delle operazioni e dei dati pubblici di una classe (operazioni e dati privati e l'implementazione dei metodi non fanno parte dell'interfaccia)

Specifica "COSA FA": ***Definisce i comportamenti utilizzabili*** dalle altre parti del programma

Keyword interface
Una classe implementa la classe astratta.

DIFFERENZE

Le principali differenze tecniche tra una classe astratta e un'interfaccia sono: le classi astratte possono avere costanti, membri, stub di metodi (metodi senza corpo) e metodi definiti, mentre le interfacce possono avere solo costanti e stub di metodi.

LEZIONE 4 : GESTIONE DELLE ECCEZIONI

Un metodo può terminare normalmente o ***sollevare un'eccezione***.
Le eccezioni vengono segnalate al chiamante che può gestire. Le eccezioni hanno un tipo e dei dati associati che danno indicazione sul problema incontrato. Le eccezioni possono anche essere definite dall'utente.

1. Sollevare le Eccezioni

Keyword **throw**, deve seguire un reference a un oggetto eccezione:

Es: if(base<=0) throw new NonPositiveBaseException();

L'oggetto eccezione:
- ha metodi e dati
- determina il tipo di eccezioni
- includono informazioni sul problema
- viene creato (spesso) quando l'eccezione viene sollevata

Effetto:
- Termina l'esecuzione del blocco codice e passa la procedura al chiamante.

2. Gestire le Eccezioni

Un'eccezione può essere gestita attraverso il costrutto try-catch:
- **try**:blocco che potrebbe sollevare eccezione
- **catch**: codice di gestione da eseguire quando si verifica l'eccezione

Ci possono essere più catch per un blocco try.

Classi Java per le eccezioni:
- Throwable, continente tutte le eccezioni lanciate dalla jvm.
  - ❖ Error: Errori che non dovrebbero mai accadere. Non possono essere catturati e creano sempre crash (Es. VirtualMachineError)
  - ❖ Exception: Eccezioni che possono essere gestite
    - ➔ **Unchecked**: sono le RuntimeException e sono lanciate dalla JVM. Non devono essere catturate (NullPointer, OutofBounds, ClassCast ecc.)
    - ➔ **Checked** Exception: eccezioni lanciate dal programma. Deve catturarla e gestirla (IO, DataFormat, IOException)

Un ramo try/catch può avere un ramo **finally** in aggiunta ai catch. E' eseguito
- sia se all'interno del blocco try non vengano sollevate eccezioni
- sia se all'interno del try vengano sollevate eccezioni, e viene eseguito dopo il catch.

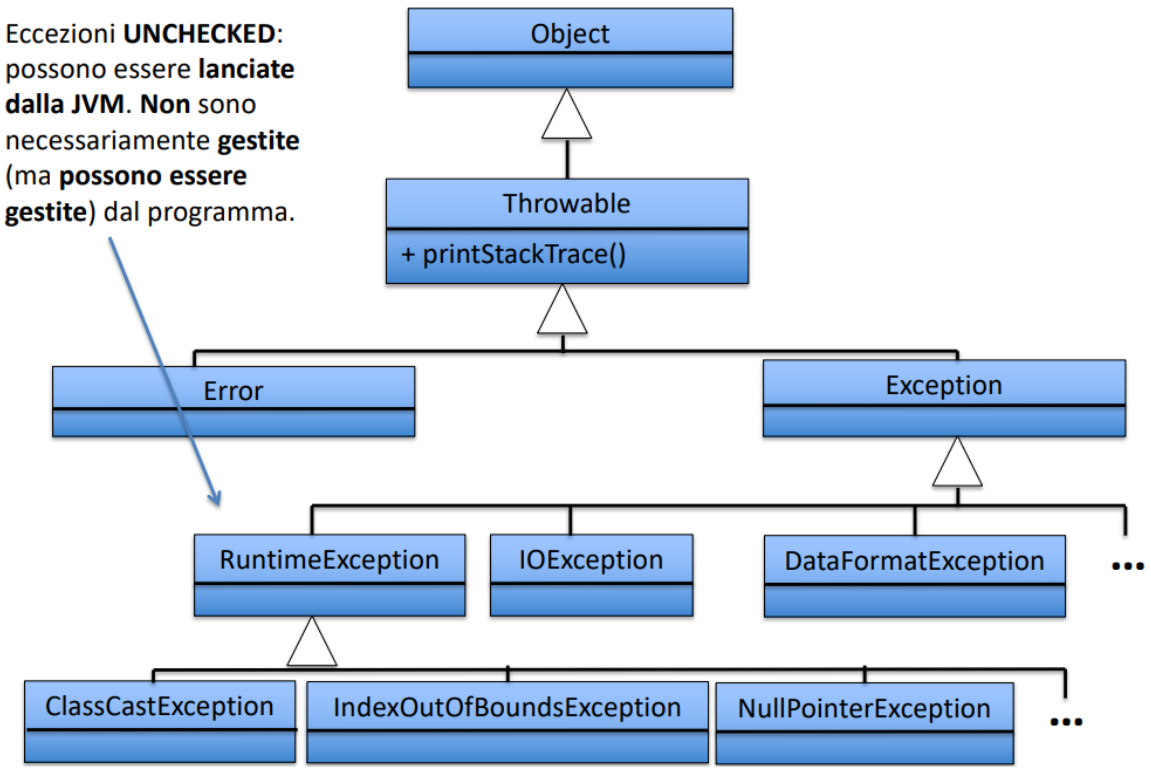
3. Definire Eccezioni

Per definire una nuova eccezione bisogna creare una classe nuova che estende exception ed implementare i costruttori

```
public class NonPositiveBaseException extends Exception{
    public NonPositiveBaseException(){ super();}
    public NonPositiveBaseException (String s){
       super(s);
    }
}
```

4. Strategie di Gestione

- **Masking**: Viene gestita l'eccezione e l'esecuzione prosegue normalmente
- **Forwarding**: L'eccezione non può essere gestita in questo momento
- **Re-throwing**: Viene catturata ,ma non può essere gestita, si prosegue con la generazione dell'eccezione di tipo differente. Come se si accumulano eccezioni e le gestiamo tutte in una.



LEZIONE 5: COLLECTION FRAMEWORK

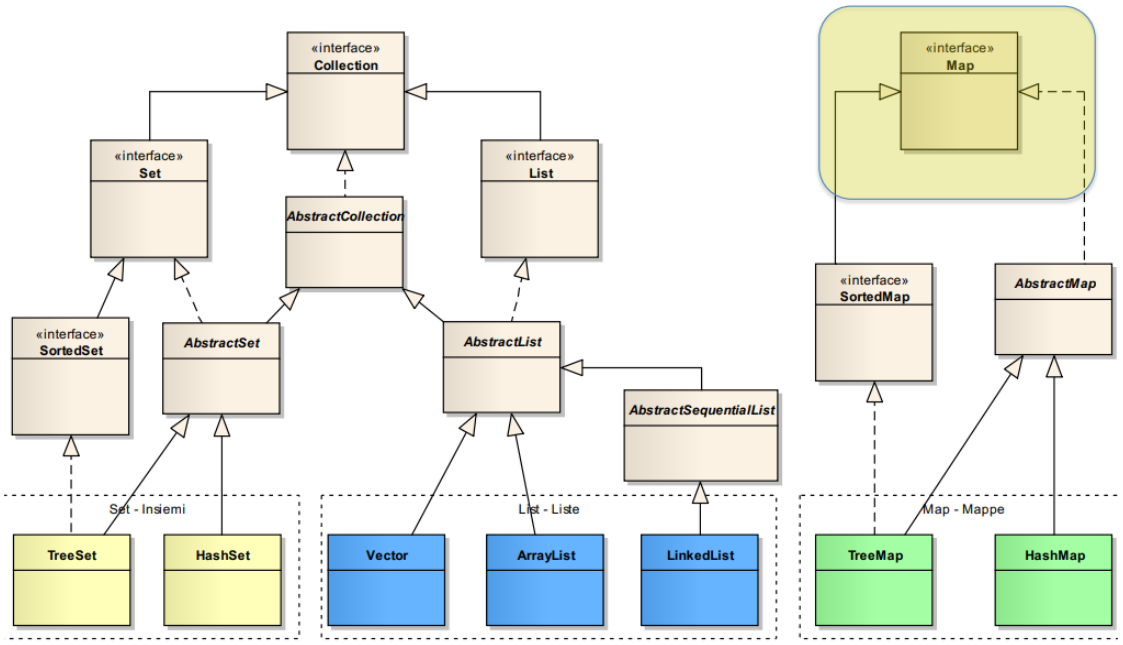
WRAPPER DI TIPI PRIMITIVI

Un ***wrapper*** di tipo primitivo è un oggetto che incapsula un attributo di tipo primitivo

- stesso comportamento del tipo primitivo
- in aggiunta può essere usato come un Object
- il compilatore trasforma in automatico i tipi primitivi in tipi wrapper e viceversa

Tipo Primitivo	Tipo Wrapper
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

COLLECTION FRAMEWORK



COLLEZIONE

Una ***collezione*** (o container) è un elemento "contenitore" di oggetti denominati elementi.

Java Collection Framework = insieme di interfacce e di classi per l'implementazione di collezioni
– Interfacce e classi astratte per rappresentare tipi di collezioni
– Implementazioni concrete pronte per essere utilizzate

La dichiarazione di tipo usa "Collection<E>"
Es: Collection<Object> c1;

Alcuni metodi dell'interfaccia:

- add(E e): aggiunge e alla collezione
- clear(): svuota la collezione
- contains(Object o): ritorna true se esiste un elemento e tale che e.equals(o)
- remove(Object o): rimuove l'elemento o (identificato tramite equals) dalla collezione
- size(): ritorna il numero degli elementi nella collezione
- toArray(): ritorna un array con gli elementi della collezione

Non dobbiamo specificare la **dimensione**.

SET

Set extends Collection: è una Collection che **non ammette elementi duplicati**
Dichiarazione di tipo usa "Set<E>"
Es: Set<Object> s1;

Non aggiunge metodi a Collection

Cambia implementazione per alcuni di essi, esempio l'add, dove controllerà che non sia già presente l'elemento.

LIST

List extends Collection: è una Collection con **elementi ordinati**
Dichiarazione di tipo usa "List<E>"
Es: List<Object> l1;

Aggiunge nuovi metodi per lavorare con un insieme ordinato(avremo index per lavorare con gli indici)
Cambia l'implementazione di alcuni metodi, come l'add e remove che saranno ordinati

MAP

NON estende Collection

• La dichiarazione: Map<K,V>
Utile soprattutto quando devo cercare, aggiungere, modificare ed eliminare gli elementi della collezione per chiave

Una chiave è **univoca**.

Ci sono vari metodi dell'interfaccia:

- clear(): svuota la mappa

- containsKey(Object key): ritorna true se esiste una chiave k tale che k.equals(key)
- containsValue(Object value): ritorna true se esiste almeno un valore v tale che v.equals(value)

- V get(Object key): ritorna l'elemento associato alla chiave key, oppure null
- Set<K> keySet(): ritorna un set con tutte le chiavi

- put(K key, V value): aggiunge value con chiave key, nel caso key esista già l'elemento esistente viene sostituito da value
- V putIfAbsent(K key, V value): se la chiave non esiste già come put e ritorna null, altrimenti ritorna il valore già associato a key nella mappa senza modificarla

- remove(Object key): rimuove l'elemento associato alla chiave key
- remove(Object key, Object value): rimuove l'elemento associato alla chiave key solamente se già presente con valore value

- size(): ritorna il numero di coppia (chiave,valore) nella mappa
- Collection<V> values(): ritorna una collection con i valori nella mappa

ORDINAMENTO

Gli elementi nelle collezioni e le chiavi possono essere oggetti qualsiasi

Gli oggetti di una classe **possono essere ordinati** se la classe **implementa** l'interfaccia **Comparable** metodo compareTo

0	Se o1 e o2 sono 'uguali'
Intero negativo	Se o1 è più piccolo di o2
Intero positivo	Se o1 è più grande di o2

Es: La classe String di Java implementa l'interfaccia Comparable.

SORTED SET

SortedMap extends Map: è una mappa con le chiavi totalmente ordinate.

Dichiarazione di tipo usa "SortedMap"
Es: SortedMap<String, Prodotto> catalogo.

Aggiunge metodi a Map:

- K firstKey(): ritorna la prima chiave
- SortedMap<K,V> headMap(K toKey): ritorna una vista della mappa contenente tutti gli elementi con chiave strettamente minori di toKey
- K lastKey(): ritorna l'ultimo valore della chiave
- SortedMap<K,V> subMap(K fromKey, K toKey): ritorna una vista della mappa contenente tutti gli elementi da fromKey a toKey (escluso)
- SortedMap<K,V> tailMap(K fromKey): ritorna una vista della mappa contenente tutti gli elementi di chiave maggiore o uguale a fromKey

IMPLEMENTAZIONI

Interfacce	Funzionalità	*Implementazioni disponibili*			
		Hash table	Array ridimensionabile	Albero ordinato	Liste concatenate
Set	Elementi non ripetuti ma eventualmente ordinati	HashSet		TreeSet	
List	Elementi ripetuti ordinati in modo posizionale		ArrayList (o Vector)		LinkedList
Map	Mappa di elementi ordinati per chiave	HashMap(o HashTable)		TreeMap	

ARRAYLIST

ArrayList ha la possibilità di implementare un Array ma non bisogna specificare la dimensione, in quanto si può stringere ed allargare.
Vettori funzionano allo stesso modo. Quando sono pieni i Vettori e devono aggiungere un elemento, duplicano la grandezza; ArrayList di metà.

LINKEDLIST

Ha le stesse funzionalità dell'ArrayList.
La LinkedList memorizza i suoi elementi in "contenitori". L'elenco ha un collegamento al primo contenitore e ogni contenitore ha un c***ollegamento al contenitore successivo*** nell'elenco.

TREESET

Essendo un Set non contiene duplicati. Mette in ordine naturale gli elementi

HASH CODE

Funzione di hash: una funzione che trasforma dei dati di lunghezza arbitraria in dati di ***lunghezza fissa***.

Indicizzare e recuperare dati di lunghezza fissa è più semplice

Gatto → funzione di Hash → Ga02h04.

Se HashCode e equals non sono specificate nella classe, vengono utilizzate quelle standard della classe Obj, che quindi creeranno un HashCode diverso se l'Obj è diverso.

HASH SET

hashcode() che ritorna il codice hash di un oggetto.
hashcode() viene utilizzato per memorizzare e recuperare in modo efficiente gli oggetti del Set.

Due oggetti uguali hanno sempre lo stesso hashCode(), ma non è vero il contrario, quindi due oggetti con lo stesso hashCode() possono essere differenti.

Per risolvere il conflitto bisogna poi fare una scansione lineare degli elementi, quindi è bene definire delle funzioni di hash code che minimizzino il numero di conflitti .

HASH TABLE (HASH MAP thread-safe)

Contiene due parametri. Quando si hanno due elementi con la stessa key, uno sovrascrive l'altro.

ITERATORE

L'iteratore è un oggetto che permette di effettuare la scansione di una Collection
metodi:

- next(): ritorna il prossimo elemento della collezione, spostandosi in avanti di una posizione
- hasNext(): ritorna true se c'è almeno un altro elemento nella collezione

FOREACH

Se il vostro codice non deve far altro che iterare su una collezione possiamo utilizzare il costrutto foreach per renderlo più compatto e nascondere la presenza di un iteratore.

Durante l'iterazione di un foreach, ***NON*** bisogna modificare la collezione di cui stiamo

VARI METODI

CompareTo() è un metodo che restituisce un numero in base al confronto che effettua. Per poterla utilizzare bisogna dichiarare nella dichiarazione classe "Comparable<Oggetto su cui si fa comparazione>.

il metodo Collections.sort() permette di ordinare una lista o comunque un oggetto non ordinato.

LEZIONE 6: I/O, STREAMS E FILE

TIPI DI STREAM

I programmi che utilizziamo usano lo stream. Uno stream è un flusso ordinato di dati in input o output. Possiamo vederli come dei canali dove facciamo fluire i dati.

Es. Input da tastiera, output su monitor.

Quando lavoriamo con gli stream, i contenuti dei file siano di caratteri o di valor

I dati nei file sono salvati come numero binari. ci sono due tipi di file:
- File di testo: il contenuto è trattato come sequenza di caratteri
- File binari: gli altri casi

Quando dobbiamo lavorare con degli stream possiamo fare riferimento a un package: java.io
Le 4 Gerarchie principali del java.io sono: reader, Writer, InputStream e OutputStream

Reader/Writer

Reader e Writer sono classi astratte per leggere e scrivere stream di caratteri. Le due classi astratte contengono tutti questi metodi, ognuno dei quali eseguirà operazioni in scrittura/lettura.

- BufferedReader/Writer usa un buffer per ottimizzare le operazioni di scrittura/lettura
- FilterReader/Writer aggiunge operazioni a uno stream esistente
- PipeReader/Writer comunicazione diretta tra un sender e un receiver
- **PrintWriter** per la stampa di caratteri e stringhe secondo un formato (SOLO scrittura)
- InputStreamReader/OutputStreamWriter per lettura e scrittura semplice da stream
    - ❖ **FileReader/Writer** per lettura e scrittura da file

InputStream/OutputStream

InputStream e OutputStream sono classi astratte per leggere e scrivere stream di valori binari:

- PipedInputStream/OutputStream comunicazione diretta tra un sender e un receiver.
- **FileInputStream/OutputStream** per lettura e scrittura da file
- ObjectInputStream/OutputStream per lettura e scrittura di oggetti
- FilterInputStream/OutputStream aggiunge operazioni a uno stream esistente. Le sottoclassi aggiungono effettivamente le operazioni.
    - **DataInputStream/OutputStream** definisce metodi per la lettura e scrittura di valori primitivi

- BufferedInputStream/OutputStream usa un buffer per ottimizzare le operazioni di scrittura/lettura
- PushbackInputStream permette di annullare operazioni di lettura (SOLO lettura)

Standard Stream

Metodi che non vanno importati nel file Java. Li troviamo già pronti.

- System.in – standard input, in genere la tastiera
- System.out – standard output, in genere il monitor
- System.err – standard output per errori, in genere il monitor

Scanner

Scanner: Semplice classe per leggere dati di tipo primitivo e stringhe. Il contenuto dello stream p formato da tokens delimitati da separatori:

- Separatore = spazi bianchi
- Token = tutto il resto, dipende dal metodo next..() invocato

Il delimitatore può essere cambiato utilizzando useDelimiter
Tipicamente lo scanner va chiuso con .close()

Ci sono vari errori:

- InputMismatchException: valore di tipo errato o out of range
- NoSuchElementException: no input
- IllegalStateException: scanner chiuso

Sono eccezioni RunTime, quindi sta a noi decidere se catturarle oppure no.

SCRITTURA DI FILE DI TESTO

Per scrivere un file di Testo abbiamo due classi principali

- FileWriter: scrittura di caratteri senza formattazione
- PrintWriter: scrittura di caratteri con formattazione

FileWriter

Viene creato un file writer: nome= new fileWriter(fileName)
filename indica un file che viene sovrascritto/creato se non esiste

Attraverso .write() possiamo scrivere ciò che ci interessa. Attraverso \n possiamo andare a capo

Chiudiamo lo stream con outputStream.close();

La scrittura può avvenire in un try, in modo tale che con catch si scriva se non si è riuscito a scrivere, e con un finally possiamo inserire il .close()

PrintWriter

Il costruttore vuole un oggetto intermedio: nome = new PrintWriter(new FileOutputStream(fileName));

Le operazioni di scrittura, a differenza dell'altro metodo, non ha bisogno di un try/catch perché non lancia eccezioni. per scrivere bisogna utilizzare .println();

La stampa formattata avviene con printf, dove all'inizio possiamo specificare come vogliamo la formattazione:

% numero (indica nome variabile) lettera (indica il tipo, decimale float ecc..)

Dopo la stringa di formattazione, inseriamo le variabili che devono contenere tale formattazione.

con un if(nome.checkError()) possiamo verificare se si sono verificati degli errori

Modifica File

Possiamo aprire un file in append. Quando creiamo la nostra output stream, specifichiamo nel costruttore:

outputStream= new PrintWriter(new FileOutputStream(fileName, true))

dove il secondo valore se

- false= file sovrascritto-> perde i propri dati precedenti
- true = apertura in append -> testo presente conservato, operazioni di modifica aggiunte in coda

FILE CLASS

File class: rappresenta un file o una cartella
file f = new File(filename)

Metodi: canRead(), canWrite(), delete(), exists(), getName(), getPath(), length().

`public boolean canRead()` Tests whether the program can read from the file.
`public boolean canWrite()` Tests whether the program can write to the file.
`public boolean delete()` Tries to delete the file. Returns true if it was able to delete the file.
`public boolean exists()` Tests whether an existing file has the name used as an argument to the constructor when the `File` object was created.
`public String getName()` Returns the name of the file. (Note that this name is not a path name, just a simple file name.)
`public String getPath()` Returns the path name of the file.
`public long length()` Returns the length of the file, in bytes.

Path

Negli esempi i file da leggere/scrivere si trovavano nella **stessa cartella** dove veniva lanciato il programma.
Possibile specificare percorsi

- Percorso assoluto (parte dalla route iniziale)
- Percorso relativo (a partire dalla prima directory)

File.separator è una costante da utilizzare per risolvere i diversi stili di separatore utilizzato da diversi os.

Cartelle

La classe File può essere usata per lavorare con le cartelle.

- list(): ritorna i file e le cartelle contenuti in una cartella
- boolean mkdir() : true → cartella creata
- boolean mkdirs(): → lungo tutto il path

LETTURA FILE DI TESTO

Per leggere un file di Testo abbiamo due classi principali

- FileReader: lettura di caratteri senza formattazione
- Scanner

FileReader

Il primo ha la stessa funzionalità del FileWriter, quindi

```
try{
        input = new FileReader("file.txt")
        while(data!= -1) ← significa che il file è finito se vale -1
        data=input.read();
}catch{eccezione
}finally{ input.close();}
```

Scanner

Così come PrintWriter, è più comodo rispetto all'altro metodo per leggere il file.

Creiamo lo scanner creando un file inputStream= newScanner(newFile(fileName)
while(inputStream.hasNextLine())
String line = inputStream.nextLine()
if(inputStream!=null) inputStream.close();

Lettura di un File CSV

Un file Comma-Separated Values (CSV) è un formato per file di testo che memorizzano liste di record. Ad esempio delle tabelle come quelle di registratori di cassa.

Abbiamo la virgola come carattere separatore

```
SKU,Quantity,Price,Description
4039,50,0.99,SODA
9100,5,9.50,T-SHIRT
1949,30,110.00,JAVA PROGRAMMING TEXTBOOK
5199,25,1.50,COOKIE
```

```
public class TransactionReader {
    public static void main(String[] args) {
        String fileName = "Transactions.csv";
        Scanner inputStream = null;
        try {
          inputStream = new Scanner(new File(fileName));
         String line = inputStream.nextLine(); // Read the header line
          double total = 0; // Total sales
          while (inputStream.hasNextLine()) {
            line = inputStream.nextLine();
            String[] ary = line.split(",");
            String SKU = ary[0];
            int quantity = Integer.parseInt(ary[1]);
            double price = Double.parseDouble(ary[2]);
            String description = ary[3];
            System.out.printf("Sold %d of %s (SKU: %s) at $%1.2f each.\n",
                                          quantity, description, SKU, price);
            total += quantity * price;
          }
          System.out.printf("Total sales: $%1.2f\n",total);
        } catch(FileNotFoundException e) {
          System.out.println("Cannot find file " + fileName);
        } catch(IOException e){
          System.out.println("Problem with input from file " + fileName);
        } finally {
          if (inputStream != null) inputStream.close();
        }
    }
```

Lavorare con File Binari

Scrittura di File Binario

FileOutputStream: scrittura valori binari
DataOutputStream: scrittura di valori tipizzati convertiti automaticamente in binari

Lettura di File Binario

FileInputStream: lettura di valori binari
DataInputStream: lettura di valori con tipo convertiti automaticamente dalla loro rappresentazione binaria

RandomAccessFile

Abilita l'accesso in lettura/scrittura in posizioni arbitrarie di un file