M.1 - PROGRAMMAZIONE AD OGGETTI

Strumento per produrre software di grandi dimensioni.
Software = entità che contengono dati e funzioni che lavorano su di essi

- perché è stato usato?

1. Modello precedente (procedurale) non è adatto a costruire software di grandi dimensioni
2. Programmazione ad oggetti diminuisce i costi di sviluppo e manutenzione. Questi ultimi sono determinati da un fattore: ERRORS/1K LINES, ove gli errori sono i difetti che possono produrmi comportamenti non voluti.

- scopo OOP?

Costruire software:
- Sicuri
- Riutilizzabili
- Flessibili
- Documentabili
- Estendibili

—> verranno ridotti i nostri per le industrie che li producono.

- problematiche legate alla programmazione procedurale:

- riuso codice limitato
- No protezione dati
- Non adatto alla decomposizione

- passaggio da procedurale ad OOP:

Modello client-server: *ho qualcuno che roga un servizio (server) e dei client che utilizzano questo servizi erogati*

1. OOP caratteristiche generali

Classe = costrutto che è in grado di mettere all'insieme , in un unica entità gli ***attributi (DATI)*** e le ***operazioni/metodi (FUNZIONI)*** *che lavorano su di essi e funzioni delegate alla costruzione degli oggetti* ***(COSTRUTTORI)***—> ***concetto astratto, categoria di oggetti***

Oggetto = *istanze materiali delle classi,* identificati da un'***identità*** (codice unico che lo identifica) e caratterizzate da uno ***stato interno*** inizializzato dal costruttore

OGGETTI CON LO STESSO STATO SONO OGGETTI DIVERSI + OGNI OGGETTI E' INDIPENDENTE

Relazione classe-oggetti: oggetti sono le istanze materiali delle classi —> quando si programma si definiscono le classi e correlazioni con oggetti sono date da **operatore NEW**, ***operatore di instanziazione***: *stanzio un oggetto a partire da una classe.*
Instanziazione: creazione di un oggetto —> oggetti creato = istanza di quella classe

—> la classe è come la definizione di un tipo: non vengono allocati dati fino a quando l'oggetto non viene creato dalla classe.

2. Concetti fondamentali che caratterizzano OOP:

A) **INCAPSULAMENTO:** ogni oggetto incapsula al suo interno delle variabili (attributi dell'oggetto, sono incapsulati all'interno della classe) che rappresentano il suo stato interno.

B) **EREDITARIETA':** *costrutto a livello di linguaggio che consente ad una classe di derivare da un'altra classe* —> la classe derivata eredita le variabili ed i metodi della classe base + aggiunge sue variabili e metodi.
—> *introdotto per riuso del codice.*
Essa si traduce con la parola EXTENDS = estendere, ereditare attributi

C) **POLIMORFISMO:** *invocazione di un metodo può portare a comportamenti diversi* in base a 3 caratteristiche:
- oggetto che compie azione vera e propria
- Tipo di parametri passati come argomento
- Contesto di esecuzione

OSS: in Java vengono definiti più costrutti con lo stesso nome, ma che portano a comportamenti differentI!!

3. Vantaggi OOP:

- *semplifica la costruzione di artefatti software di grandi dimensioni in un ambiente cooperativo:* persone diverse possono sviluppare classi diverse —> incapsulamento

- *Semplifica la gestione del codice:* dati sono incapsulati quindi gli errori molto spesso possono essere trovati dentro oggetto che contiene la variabile con un valore che non dovrebbe avere, cioè gli errori sono già incapsulati dentro una classe

- *Supporta lo sviluppo incrementale:* meccanismo ereditarietà

4. Svantaggi OOP:

- *Richiede la capacità di pensare ad oggetti*, che hanno stato interno e operazioni
- *Design relativamente complicato*
- *Vantaggi sono ed emergono con maggior dimensione del codice*

M.2 - INTRODUZIONE AL LINGUAGGIO JAVA

Linguaggio di riferimento per sviluppo di applicazioni industriali

1. Caratteristiche del linguaggio:

- ***Indipendenza dalla piattaforma***: scrivi una volta, esegui il codice ovunque
- ***Linguaggio ad oggetti puro***
- ***Linguaggio strettamente tipizzato***: tutto ha un tipo, controllato dal compilatore

- ***Ha sistemi di garbage collection automatica***: ogni volta che alloco memoria, non è necessario deallocarla usando una free( )

Programmi compilati diversi da interpretati:

Programmi compilati (linguaggio C): parte dei file sorgente che vengono compilati da compilatori specifici per SO—> producono file oggetto eseguibili che il linker trasforma in eseguibili specifici per SO
Programmi interpretati (Java): sorgenti Java vengono compilati da compilatore Java C, producono dei file.class che vengono eseguiti su SO tramite Java virtual machine

Rapporti fra programmi, file e classi:

Programma = gruppo si package
Package = raggruppamento di classi —> + file
File = può contenere 1 o più classi, in particolare una sola classe pubblica (ha nome che inizia con lettera maiuscola e file deve avere lo stesso nome)
Classe = ha lo stesso nome del file ad essa associato

Metodo MAIN:

In Java non esistono funzioni, ma solo metodi all'interno di classi
—> metodo iniziale:

public static void main (String[ ] args) {

}

2. Concetti base linguaggio Java:

- **Commenti:** 2 notazioni
Ispirata al C **/* …. */**
Commenti su singola linea **//…**

- **Blocchi di codice:** molto simile al C + blocchi identificati ad inizio di for, if { }

- **Strutture di controllo:** come il C
If-else
Switch-case
Do-while
Continue
For
Break
While

- **Passaggio di parametri:** in C avviene per valore o per riferimento (ptr), mentre in Java avviene sempre e solo per valore = riferimento all'oggetto

- **Tipi primitivi:** 3 tipologie
A) Interi: *byte = 8 bit*
Char = 16 bit x caratteri

Short = 16 bit x interi
Int = 32 bit
Long = 64 bit
B) N° con virgola:
Float = 32 bit
Double = 64 bit —> x default vengono preferiti da Java

C) Altri:
Boolean: contiene v/f
Void : nessun valore

- **Costanti:** in C abbiamo la parola CONST, in Java abbiamo ***FINAL*** + identificare con scrittura maiuscola e scritte ad inizio classe

- **Operatori:** *aritmetici, relazionali, bit a bit, logici, assegnamento, incremento + operatori logici (if) funzionano solamente con valori booleani*

- **Convenzioni Coding:** *programmando ad oggetti software di grandi dimensioni, devo seguire delle convenzioni:*

Classe pubblica con nome maiuscolo per ogni parola che la compone
Variabile costante ad inizio classe ed identificate con tutte tutte maiuscole
Variabile inizia con minuscola
Indentazione codice: CMD+A , CMD-I

3. Le stringhe

In C sono dei vettori di caratteri = spazio in memoria dove vengono memorizzate sequenze di caratteri.
In Java sono ***istanze di una particolare classe chiamata String + sono immutabili:*** non possono essere modificate —> nell'istante in cui la definisco, questa non può più essere modificata!

- **Modalità inizializzazione:**

1. **Breve:** *String s1= "ciao"; —> dove vanno ?* salvate in memoria (in pula di stringhe) statica e stringhe con = valore non vengono risalvate

2. **Modalità corretta/modalità dinamica:** *String s3 =* ***NEW*** *String ("nicole"); —>* modalità corretta perché le stringhe sono oggetti e come tali si inizializzano utilizzando parola NEW —> *dove vanno ?* Finiscono nell'heap

- **Concatenazione stringhe:**

Se devo concatenare frequentemente uso una classe: **stringbuffer**

Ex) String s1= "pippo";
String s2 = "è bello";
System.out.println(s1 + " " + s2);

- **Hanno diversi metodi:** idea—> ho un oggetto e chiedo servizio a questo oggetto

A) Il più comune è sapere quanto sono lunghe: *System.out.println(s1.lenght( ));*

B) Carattere che compare al posto i-esimo: *System.out.println(s1.indexOf( ));*
C) Altri —> Java Ipa Classe: manuale dell'oggetto che sto usando

- **Hanno stesso metodo:** le stringhe in considerazione puntano allo stesso oggetto in memoria

Ex) uso equals: ***if (s1.equals(s2))*** con s1,s2 inizializzate in modalità dinamica
—> equals mi restituisce vero se i 2 oggetti hanno lo stesso contenuto

4. Array

Sono una *sequenza ordinata di variabili dello stesso tipo* che possono essere accedute dallo stesso tipo + possono contenere tipi primitivi e riferimenti ad oggetti + definiti in dimensione al tempo di creazione.

- ***dichiarazione:*** **int v[ ]** = vettore di interi oppure **int [ ] v** = riferimento a vettore di interi

Gli array sono oggetti e *devono essere creati usando* ***operatore NEW*** che costruisce l'oggetto e lo memorizza nell'heap (memoria dinamica)
—> oggetto vettore di interi :
Int [ ] v = new int [16] (v è un vettore di 16 interi)
Con **v.** Ho *accesso a tutti i metodi dei vettori*

- ***inizializzazione statica:*** int [ ] v = {1,2,3,4,4,6};

- ***operazioni con gli array:*** non esiste l'aritmetica dei puntatori
Java controlla i bordi degli array

Riferimento ad un array non è un puntatore la primo elemento, ma un riferimento all'oggetto vettore.

+ **nuovo costrutto di tipo loop: For (Type var : set_expression)** dove set_expression = array (indice implicito) oppure classe che implementa Iterable
+ **Array multidimensionali:** valgono le cose viste in C —> **Person [ ] [ ] table = new Person [2] [3]; Table [0] [2] = new Person ("Mary"); +** le righe possono essere interscambiate.

5. Visibilità costruttori, getters e setters

Classi = rappresentano una *categoria di oggetti* + *contengono dei metodi*
OSS: COMPONENTI FONDAMENTALI DI UNA CLASSE:
- Definizione della classe
- Definizione dei suoi attributi
- Costruttori = metodo che contiene operazioni che vogliamo eseguire immediatamente appena gli oggetti sono stati creati —> se non definito, un costruttore di default senza parametri viene definito
- Getter/Setter
- Nostri metodi
- toString

Gli ultimi quattro punti possono essere creati in modo automatico :
- source —> generate constructor using fields
- Generazione getters/Setters
- Generazione di ToString( )

Metodo ToString ( ):
È comodo per poter ottenere una rappresentazione testuale degli oggetti.

Oggetti = istanze materiali, *particolari rappresentazioni di quella classe* —> contengono attributi a cui assegnano un valore, ogni oggetti ha un particolare stato interno e ciò che differenzia gli oggetti tra loro è lo stato interno.
Gli attributi sono incapsulati nell'oggetto ed esposti all'esterno attraverso metodi

- **incapsulamento:** a livello di linguaggio è possibile determinare due livelli di visibilità di attributi e metodi

Descrittori di visibilità:

a) **Public:** tutti possono fare tutto
b) **Private:** attributi sono accessibili solo all'interno della classe stessa—> uso una famiglia di metodi **getter** e **setter**, che *rendono accessibili gli attributi sia in lettura che scrittura*

Getter = *prelevano, forniscono all'esterno il valore degli attributi —> generalmente è senza parametri*
Setter = *sentano gli attributi —> un solo parametro = stesso tipo dell'attributo che voglio settare*

c) **Default:** non c'è scritto nulla —> sono accessibili alla classe stessa, nelle classi dello stesso package e sottoclassi dello stesso package
d) **Protected:** uguale a default + visibile dalle sottoclassi anche in altri package

6. Creazione e distruzione oggetti

- **Overload dei metodi:** importante nell'ambito dei costruttori, esso significa: **scrivere un nuovo metodo con stesso nome, ma parametri diversi**

I metodi possono avere parametri.
In generale in una classe potrebbero esistere due metodi con lo stesso nome ma firme diverse, firma in Java è costituita dal nome del metodo e dalla lista ordinata dei tipi dei parametri che questo metodo riceve —> viene invocato un metodo con un certo nome (esistono più metodi con stesso nome), viene invocato da Java QUELLO alla cui lista dei parametri corrisponde l'invocazione.

- **cosa definisce un oggetto:**

A) **Classe: determina la sua struttura in termini di attributi e metodi**

B) **Stato: valore degli attributi, variabili interne**

+ **tutti gli oggetti creati hanno un identificatore unico —>** come lo vediamo: elimino metodo ToString ( ) e faccio partire il programma
+ **ha anche 0/1 o più riferimenti che si riferiscono a lui**

Creazione di un oggetto:

- **operatore new:** ritorna un riferimento assegnato alla variabile che usiamo —> chiama metodo costruttore della classe che vogliamo creare: non hanno tipo di titolo e hanno stesso nome della classe + possono esistere parametri con operatori diversi = overloading
- **OSS:** se il costruttore non è definito all'interno di una classe, esiste un costruttore du default senza parametri automaticamente costruito da Java

Parola chiave this:
Può essere utile nei metodi per distinguere tra gli attributi all'oggetto e le variabili locali + modo in Java per dire "me stesso" perché la classe lo usa per riferirsi a se stessa

Distruzione oggetti:
In C: malloc ( ) —> free ( )
In Java: Garbage Collection —> disalloca la memoria di tutti gli oggetti che nn hanno riferimenti attivi. Non dobbiamo preoccuparci di dislocare la memoria dinamica

Operatore ==/!= tra due riferimenti:
Cerco di verificare se i 2 riferimenti puntano allo stesso oggetto

Utilizzo del punto:
Accedo alle componenti interne di un oggetto
Ex) System.out.println ("Hello world!") —> prendo oggetto system che ha attributo interno che si chiama out —> oggetto out ha metodo interno println che prende una stringa come parametro e fa qualcosa:

- **System:** *classe messa a disposizione dalla virtual machine appartenete a Java.long, che mette a disposizione un'interfaccia verso il sistema*
- **Out:** *oggetto che rappresenta il terminale di uscita —> essendo un oggetto ha dei metodi*
- **Println ( ):** *metodo con cui posso stampare qualcosa all'esterno*

7. Attributi e metodi statici

Usati per memorizzare tutte le istanze di un oggetto. Esistono anche quando nessuno oggetto è stato istanziato

8. Classi wrapper

Versione ad oggetti dei tipi primitivi —> essi erogano in convezione tra tipi primitivi, stringhe, oggetto

PRIMITIVE TYPES	WRAPPER CLASS
Boolean	Boolean
Char	Character
Byte	Byte
Short	Short
Int	Integer
Long	Long
Float	Float
Double	Double
Void	Void

- **Auto-boxing:** conversione automatica che il compilatore Java esegue fra tipi primitivi e corrispettivi oggetti wrapper

- **Auto-unboxing:** processo opposto all'auto-boxing

9. Packages

Motivazione: classi sono elementi di modularizzazione più efficaci delle funzioni del C: contengono dati e funzioni MA sono relativamente piccole —> x modulatizzare il codice usando unità più grandi, sono stati introdotti i package.

Package = *serie di definizioni di classi* —>sottocartella della cartella sorgente, che contengono delle definizioni di classi.

COSA FA?
- definire nuovi ambirete di visibilità
- Possibilità di avere classi con lo stesso nome in pakcage diversi senza creare conflitto
- Identificato da un nome interno ad una struttura gerarchica: Java.lang , Java.util , Java.sql..
- Quando scrivo i package c'è convenzione di avere nomi unici: it.unimore.mypackage

Definizioni = all'inizio delle classi che appartengono al pakcage: introduzione della definizione di pakcage
—> Package packageName

Uso: per usare classe classe apparente ad un package devo importarla
—> import packageName.ClassName

OSS: se due package definiscono una classe con lo stesso nome, le due classi non possono essere importate insieme. Se dovessimo usare contemporaneamente due classi con lo stesso nome appartenetene a package diversi, *ne importo una e uso l'altra usando il nome completo*

Ex) import Java.sql.Date;
Date d1 = new Date ( );
Java.util.date d2= new java.util.Date ( );

Questione della visibilità delle classi (pubbliche/private) + dover definire interfaccia verso l'esterno sono valide anche per i package:
- regole di visibilità si applicano anche ai pakcage
- Interfaccia esterna package = set di classi pubbliche contenute nel package

—> suggerimenti:
- considero i package come entità di modularizzazione
- Minimizzo il numero di classi, attributi, metodi visibili dall'esterno e teniamo il resto incapsulato nel package

MODULO 3.

1. Ereditarietà

Possiamo avere uno strumento a livello di linguaggio che ci acconsenta di accorpare codice comune in un unico punto tale che le modifiche non si propaghino in tutte le classi che abbiamo scritto.

- **Motivazione:** frequentemente una classe è solamente una modifica di un'altra classe —> ereditarietà consente di minimizzare ripetizione del codice comune

Localizzazione del codice:
- sistemare un bug in una classe base —> sistema classi figlie
- Aggiungo una funzione in una classe base —> aggiungo anche nelle classi figlie
- Si riducono possibilità di implementazione inconsistenti

In programmazione ad oggetti com'è l'ereditarietà:

Classe può essere un sottotipo di un'altra classe
Classe che eredita contiene tutti gli attributi e metodi dalla classe da cui ha ereditato
Classe che eredita può definire attributi e metodi aggiuntivi
Classe che eredita può sovrascrivere la definizione di metodi esistenti

Con termine **EXTENDS** la sottoclasse estende sé stessa dalla classe padre

2. Ereditarietà e costruttori

Dal momento che ogni sottoclasse contiene un'istanza della classe genitore, che deve essere inizializzata prima che venga inizializzata la seconda —> compilatore java chiama costruttore di default della classe genitore ogni volta che la classe figlia viene creata, questa chiamata viene inserita come prima istruzione nel costruttore delle classi figlie.

- ***This:*** riferimento classe/oggetto corrente
- ***Super:*** riferimento alla classe genitore
- ***Super ( ):*** chiama costruttore di default della classe
- ***Super (params):*** chiama costruttore specifico della classe

Metodo corretto di procedere? ***Definire un costruttore all'interno di TUTTE le classi*** (genitori e figlie—> in queste ultime la prima cosa da fare è chiamare il costruttore della classe genitore)

Come vengono costruiti gli oggetti figli? Esecuzione dei costruttori procede dall'alto al basso lungo la catena dell'ereditarietà —> quando metodo della classe figlia è eseguito (incluso il costruttore), la superasse è inizializzata.
IDEA: se una sottoclasse confine un'istanza delle sue classi genitrici, affinché la sottoclasse sia inizializzabile devono essere già inizializzata tutta la catena dei suoi antenati

DYNAMIC BINDING E POLIMORFISMO —> DOMANDA SPESSO FATTA ALL'ORALE
Casi attraverso i quali si può avere polimorfismo?
A) metodo con stesso nome ma parametri diversi —> **overloading**
B) Ereditarietà, **overide dei metodi —> =** sovrascrivere un metodo esistente di una super-classe in una sottoclasse

3. Classe oggetto

Ogni classe deriva da un antenato comune: classe contenuta nel package **java.lang.object, che è la classe più generica di tutte** —> rende elegante il linguaggio + contiene servizi basici che è bene che le classi abbiano:
- **toString ( ):** ritorna la rappresentazione in stringa di un oggetto
- **Equals (Object o):** ritorna v/f se contenuto dell'oggetto è uguale al contenuto dell'oggetto passato come parametro
- **Clone ( ):**

TUTTE LE CLASSI HANNO METODI APPARTENENTI AD OBJECT!

4. Upcasting e down casting

Casting: Java è un linguaggio estremamente tipizzato —> ogni variabile ha un tipo!! Servono per controllare che qualcuno non inserisca un tipo NON corretto in una variabile :

Ex)
Float f : - corretto: f=4.7;
- sbagliato: f= “hello!”:

Car c : - corretto : c = new Car ( );
- sbagliato: c= new String ( );

MA ABBIAMO ALCUNI CASI PARTICOLARI:
Class car ( );
Class SDCar extends Car { };

Car c1= new Car ( );
SDCar c2 = new SDCar ( );

Car c3= new SDCar ( ) ; —> c3 è riferimento di tipo car che uso per manipolare un oggetto di tipo SDCar
***Riferimento* =** variabile simile ad un puntatore che contiene l’indirizzo di un oggetto (memoria dinamica) creato dall’operatore new —> ci dice quali sono i metodi che possiamo chiamare sull’oggetto.

Upcasting e down-casting: possibilità di cambiare il riferimento ad un tipo di un dato oggetto —> oggetto resta lo stesso, ma posso cambiare il riferimento.

1) UPCASTING: usare riferimenti più generici

Class Car { };
Class SDCar extends Car { };
Car c= new SDCar ( ); —> upcasting: assegnamento di un riferimento più generico rispetto all’oggetto creato.
-caratteristiche:
A) Dependable: possiamo dipendere da lui, è affidabile, non crea problemi —> è sempre vero che un SDCar è una Car too
B) Automatico: non ci serve parentesi per forzare il casting —> car c= new SDCar ( );

2) DOWNCASTING: usare riferimenti più specifici
—> operatore **ISTANCEOF:** keyword che dice quando una variabile **è** membro di una classe o interfaccia, cioè quando ha i metodi definiti in quella classe/interfaccia

5. Classi astratte e interfacce

Metodi astratti:
in Java è possibile dichiarare un oggetto senza implementarlo : person p;
In modo analogo è possibile dichiarare un metodo senza fornirgli implementazione (corpo del metodo è mancante) : public **ABSTRACT** void draw (int size) —> metodo non implementato

Classe astratta:
Classe che contiene uno o più metodi astratti —> la presenza di un metodo astratto, implica che la classe diventi astratta
Di conseguenza :
- non posso usare l’operatore new perché ha metodi indefiniti
- Possono essere estete da figlie che rimangono astratte

Specializzare un’interfaccia: aggiungere nuovi metodi —> le interfacce possono essere implementate parzialmente in classi astratte

Ereditarietà multipla: in Java una classe può estendere soltanto un'altra classe e può implementare più interfacce (set di metodi) che vanno intese come funzionalità che si aggiungono a quelle della classe esistente —> soluzione: interfacce!!

Class Application EXTENDS Frame IMPLEMENTS ActionListener, KeyListener { …. };

MODULO 4.

1. Framework Collection

JCF è un gruppo di classi ed interfacce che implementano strutture dati comuni e riutilizzabili. *È un package (package.java.util) che mette a disposizione:*
- Interfacce che definiscono funzionalità di base
- Classi astratte
- Classi concrete erogano le funzionalità vere e proprie
- Algoritmi (java.util.Collections) contiene metodi statici: utilità e algoritmi da applicare alle collection

Concetti fondamentali :

a) **Array ri-dimensionabile:** *classe che contiene array che ci offre come servizio l'illusione che questo array si possa ridimensionare* in base a quanti oggetti inseriamo all'interno. Condizione di *Overflow* si verifica quando il vettore è pieno —> vettore aumenta la dimensione per contenere altro elemento

—> *complessità computazionale* ***O(n)***

b) Lista linkata: lista di elementi che parte da un riferimento noto come un puntatore alla testa della lista che contiene payload poi legato ad un link=puntatore all'elemento successivo della lista —> lista dove gli elementi sono separati in memoria e legati da link

—> *complessità computazionale* ***O(n)***

c) Albero bilanciato: mantiene gli elementi ordinati e bilanciati —> Albero è tale se e solo se il numero di sotto-nodi a sinistra e destra del nodo indicato differisce al massimo di un elemento

—> *complessità computazionale* ***O(log(n))***

d) Hash table: struttura dati che ci consente di eseguire operazioni che non dipendono dal numero di elementi

—> *complessità computazionale* ***O(1)***

—> come funzionano: approccio tradizionale di un vettore si riempirebbe la lista appena compaiono nuovi elementi dall'inizio alla fine, MA io voglio mettere gli elementi in vettore in posizioni sparse e precise: ***funzione di hashing*** associa ad un set di informazioni di partenza un valore di destinazione. ***Collisione: 2 elementi nello stesso slot del vettore —> java riferisce allo slot una lista.***

Complessità computazionale = complessità media delle operazioni che si compiono su una struttura dati di un dato tipo.

2. Interfaccia Iterabile e Collection

Iterable = interfaccia capostipite della gerarchia che viene specializzata (aggiunti metodi in interfaccia Collection) non si usa mai esplicitamente. **Contiene solo un metodo** che si chiama ***Iterator( )*** che ritorna un oggetto iteratore. *Iterator ( ) è un'interfaccia con 3 metodi:*

- ***boolean hasNext( ):*** esiste prossimo elemento?
- ***Object next( ):*** ritorna elemento successivo

- ***Void remove( ):*** rimuove un elemento dalla collezione

Interfaccia Collection = *interfaccia che rappresenta un gruppo di elementi* (riferimenti a quegli oggetti) *e NON specifica se questi sono non/ordinati oppure non/duplicati*

Costruttori principali: —> l'hanno anche le figlie di Collection
- Collection ( )
- Collection (Collection c) —> le Collection hanno un costruttore per generare una Collection partendo da un'altra Collection

3. Interfaccia List

È una particolare figlia di collection ( ), che consente la *duplicazione degli elementi*, *conserva ordine di inserimento, in cui è possibile inserire elementi in posizioni arbitrarie ed in cui gli elementi possono essere acceduti per posizione.*

Ha metodi aggiuntivi che specializzano ulteriormente Collection

Inzializzazione:
List<Integer> l = new ArrayList <Integer>( ) ;
l.add(14);
l.add(73);

List <Integer> l= new ArrayList ><Integer>(Arrays.aslist (14,73));

4. Interfaccia Set

Non contiene metodi addizionali rispetto a Collection ( ), ma *NON accettano elementi duplicati.*
Uso più comune: eliminano i duplicati dalle liste.

Implementazioni:
- **HashSet** implementazione di Set basata su Hashtable che è molto veloce, ma non preserva ordine di inserimento
- **LinkedHashSet:** ha stesse funzioni di hashset, ma preserva l'ordine di inserimento
- **TreeSet:** versione che implementa algoritmo di ordinamento interno + mantiene i dati ordinati al suo interno —> in particolare:

TreeSet ( ) : per tipi semplici è necessario implementare interfaccia Comparable per avere elementi ordinati, altrimenti ordinati secondo ordine naturale
TreeSet (Comparator c) : Comparator è l'ordine che viene utilizzato dal TreeSet

5. Interfaccia Queue

Collection progettate per mantenere elementi prima del processing —> aggiunge metodi addizionali per inserimento ed estrazione + *aggiunge il concetto di testa e coda. Tipicamente si aggiunge da una parte e si preleva dall'altra.*
- **metodi:**

1) Boolean add(object o) —> aggiunge elemento

2) Prelevano elemento da coda senza eliminarlo + ritornano un errore in modalità C (null)
Object peek ( )
Object poll ( )

3) Prelevano elemento da coda eliminandolo ed errore = eccezione
Object element ( )
Object remove ( )

- **implementazioni:**

LinkedList implementa List e Queue + dati escono seguendo politica FIFO
PriorityQueue = coda con meccanico di ordinamento al suo interno —> con interfaccia Comparable è possibile definire il criterio di ordinamento da applicare agli elementi della lista

- **relazione PriorityQueue e TreeSet?**

A) **Analogie:** entrambi offrono operazioni di aggiungere, rimuovere, creare elementi con complessità logaritmica O(log(n)) + mantengono elementi ordinati al loro interno usando qualche ordine definito nell'interfaccia Comparable
B) **Differenze:** TreeSet è un set = non ammette duplicati, mentre PrioritàQueue è una Queue che permette duplicati + sono metodi che se usati con interfaccia canonica offrono dati sempre ordinati, se cerco di iterare sulla struttura dati: TreeSet ordine garantito a differenza di PriorityQueue.

6. Interfaccia Map

Non deriva da iterabile + non tratta gli elementi singoli ma coppie di elementi associativi **(Chiave,Valore)** *dove le chiavi devono essere uniche, ed entrambi gli elementi devono essere oggetti.*

- **Costruttori principali:**

Map ( ) —> crea una mappa vuota, non accetta alcun parametro
Map (map m)

- **Metodi principali:**

- Object put(Object key, Object value) —> creazione di una mappa con data chiave e dato valore
- Object get(Object key) —> ritorna il valore key
- Object remove(Object key) —> ritorna la coppia (K;V) identificata dal valore di Key passato come parametro
- boolean containsKey(Object key) —> chiedo alla mappa se contiene Key
- boolean containsValue(Object value) —> chiedo alla mappa se contiene Value
- public Set keySet() —> darmi il set di chiavi
- public Collection values() —> darmi collection di valori
- int size() —> quante coppie contiene la mappa
- boolean isEmpty() —> la mappa è vuota o piena?
- void clear() —> svuotare la mappa

- **Implementazione:**

1) HashMap ( ) — mappa basata su tabelle Hash, metodo più veloce e più usato anche se viene perso l'ordine degli elementi
Get/set avvengono in tempo costante + è possibile definire un costruttore

Map<String, Integer> m = new HashMap<String, Integer>();

2) LinkedHashMap ( ) — analogo a quello sopra, ma NON viene perso l'ordine degli elementi
3) TreeMap ( ) —> implementa SortedMap (figlia di Map) + i dati sono ordinati secondo un certo criterio — poco utilizzato

7. Iteratori

Relazione iteratori e Collections:
È insicuro modificare (aggiungere o togliere elementi) da una collection mentre stiamo iterando su di essa

Ex)
```
List<Double> l = new LinkedList<Double>( Arrays.asList(10.8, 11.1, 13.2, 30.2));
int count = 0;
for (double i : l) {
if (count == 1) l.remove(count); —> posizionati sul secondo elemento, chiamo remove per rimuovere secondo elemento della lista
if (count == 2) l.add(22.3); —> 3° elemento aggiungo altro double
count++;
} // Wrong! We modify the list while iterating
```

Problema si risolve con ***iteratori, USATI IN MODO ESPLICITO PER MODIFICARE LE COLLECTION MENTRE LE STIAMO SCORRENDO***
2 INTERFACCE:

A) ***ITERATOR:*** unidirezionale e ci consente di rimuovere elementi, ma ha solo un metodo next per andare avanti

```
List<Double> l = new LinkedList<Double>( Arrays.asList(10.8, 11.1, 13.2, 30.2));

int count = 0;

for (Iterator<Double> i = l.iterator( ); i.hasNext( );) {
double d = i.next( );
if (count == 1) i.remove( );
count++;
}
```

B) ***LISTITERATOR:*** bidirezionale e consente di togliere e aggiungere elementi

```
List<Double> l = new LinkedList<Double>( Arrays.asList(10.8, 11.1, 13.2, 30.2));

int count = 0;

for (ListIterator<Double> i = l.listIterator( ); i.hasNext( );) {
double d = i.next();
if (count == 1) i.remove( );
if (count == 2) i.add(22.3);
 count++;
}
```

OSS: FOR ( ) E ITERATORI

```
/* C style */
for (int i = 0; i < pl.size; i++)
System.out.println(pl.get(i))

/* Java style */
for (Person p : pl)
System.out.println(p);

/* Iterator style */
for(Iterator<Person> i = pl.iterator( ); i.hasNext();) {
```

```
Person p = i.next( );
System.out.println(p);
}

/* While style */
Iterator i = pl.iterator( );
 while (i.hasNext( ))
System.out.println((Person)i.next( ));
```

8. Ricerca e ordinamento

Algoritmi di ordinamento —> 2 classi fondamentali:
Java.util.Collections che fornisce delle funzionalità frequenti da usare con le Collections
Java.util.Arrays che fornisce delle utilità da usare comunemente con degli array
Entrambe utilizzano metodi statici

- **Java.util.collections —> metodi principali:**

Sort ( ) —> implemetazione del merge sort, che implementa la lista
binarySearch ( ) —> ricerca binaria applicabile solo su liste ordinate = riordinare collection
Shuffle ( ) —> rimescola il contenuto della lista
Reverse ( ) —> rigira la lista
Rotate ( ) —> sposta a destra o a sinistra gli elementi di una lista dato un certo numero passato come parametro
Min ( ), max ( ) —> trovano massimi e minimi dentro collection di elementi

- **quali sono i criteri per ordinare una lista di oggetti?**

Uso 2 interfacce:

A) Comparable: creata e definita per essere introdotta all'interno della classe di cui vogliamo rendere confrontabili gli elementi

è implementata per i tipi comuni di lingua nei pacchetti java.lang e java.util. Per esempio:
- Gli oggetti stringa sono ordinati lessicograficamente
- Gli oggetti data sono ordinati cronologicamente
- Numero e sottoclassi sono ordinati numericamente

Esempio:

```
class Person implements Comparable<Person> {
  protected String name;
  protected String lastname;
  protected int age;
public int compareTo(Person p) {
// order by surname
cmp = lastname.compareTo(p.lastname);
if(cmp == 0)
      // if equal surnames, order by name
cmp = firstname.compareTo(s.firstname);
return cmp;
}
```

B) Comparator: scritta per essere inserita in una classe vuota, che implementa comparato —< confronto due elementi

Entrambe le interfacce ritornano un intero:
- *Se è >0:* l'oggetto corrente è prima dell'oggetto parametro
- *Se è < 0:* l'oggetto corrente è dopo il parametro
- *Se è =0:* stesso posto del parametro

MODULO 5.

1. Generics concetti generali

Generics è stato aggiunto in Java 5 per fornire il ***controllo del tipo in fase di compilazione*** e **rimuovere il rischio di ClassCastException** che era comune durante il lavoro con le classi di raccolta.
• L'intero quadro di raccolta (JCF) è stato riscritto per utilizzare i generici per la sicurezza dei tipi.

Cosa c'era prima dei Generics?
List fruitList = new ArrayList( ); —> si perdeva tipizzazione del linguaggio e potevo inserire correttamente frutto o anche vegetale, esponendomi ad errore runtime = classCastException. **Il compilatore non sa che fruitlLst punta ad un oggetto che può contenere solo frutti.**

Soluzione programmatore inesperto —> Codice duplicato
fruitList.add(new Fruit("Apple"));
fruitList.add(new Vegetable("Carrot"));

Fruit f;
f = (Fruit) fruitList.get(0);
f = (Fruit) fruitList.get(1);
// Runtime Error! (java.lang.ClassCastException)
Compiler doesn't know that *fruitList* should only contain fruits
We could make our own fruit-only list class:
class FruitList {
void add(Fruit f) { ... }
Fruit get(int index) { ... }
Fruit remove(int index) { ... } ...
}

Soluzione —> generics
class GenericList<**T**> {
void add(**T** element) { ... }
T get(int index) { ... } T
remove(int index) { ... }
...
}
T= classe utilizza tipi generici

Esempi di classi che utilizzano tipi generici:
- **ArrayList:**
- **Interfaccia Comparable:** public interface Comparable<T> {
  public int compareTo(T o);
}
- **Interfaccia Comparator:** public interface Comparator<T> {
  public int compare(T o1, T o2);
}

RIASSUMENDO:

// Raw types: Evil

```
List l = new ArrayList( );
l.add(new Fruit( ));
l.add(new Vegetable( )); // Succeeds but should not
...
for (Object o : l) {
Fruit f = (Fruit) o;
// Downcast eventually leading to run-time error
...
}

// Generic types: Good
List<Fruit> l = new ArrayList<Fruit>( );
 l.add(new Fruit( ));
l.add(new Vegetable( )); // Compile-time error ...
for (Fruit f : l) { ...
}
```

2. Generics bounded wildcards

Vedi PPT con esempi

3. Generics metodi

Metodi che introducono i loto parametri che hanno visibilità all'interno del metodo in cui sono stati dichiarati.

Esempio:
static <T> void fill(List<? super T> list, T obj); —>> usato per sostituire tutti gli elementi dell'elenco specificato con l'elemento specificato.
static void reverse(List<?> list);
static void shuffle(List<?> list);

Uso o meno delle wildcards con metodi generici?
Regola empirica: i metodi generici consentono di utilizzare parametri di tipo per esprimere dipendenze tra i tipi di uno o più argomenti per un metodo e / o il suo tipo restituito. Se non esiste tale dipendenza, non si dovrebbe usare un metodo generico.

Scrittura:

```
// Java API
interface Collection<E> {
public boolean containsAll(Collection<?> c);
public boolean addAll(Collection<? extends E> c);
 }

// Alternative legitimate version
interface Collection<E> {
public <T> boolean containsAll(Collection<T> c);
public <T extends E> boolean addAll(Collection<T> c);
}
```

4. Generics implementazione

Code erasure: dopo la compilazione il compilatore scarta tutti i generics e la virtual machine non li vede mai —> simile al preprocessore del C con direttive #
Come funziona questo metodo?
- vengono lanciate informazioni sul tipo tra parentesi angolati fuori. Per esempio: <String>
- Gli usi delle variabili di tipo sono sostituiti dal tipo più generico che può assumere (solitamente Oggetto)
- I cast vengono inseriti per preservare la correttezza del tipo

- ***Vantaggi metodo:*** compatibilità con le versioni precedenti viene mantenuta, quindi è ancora possibile usare librerie non generiche
- ***Svantaggi metodo:*** non puoi sapere quale tipo viene usato da una classe generica in fase di esecuzione

MODULO 6.

1. Eccezioni concetti generali

Eccezioni Java sono un meccanismo di controllo degli errori, e gli sviluppatori possono seguire per risolvere :
- ***funzione torna un particolare valore*** —> devo ricordarmi i codici di ritorno delle funzioni —> difficile + solo il chiamante può intercettare gli errori!
- ***Interrompere l'esecuzione*** —> codice non riutilizzabile, quindi va evitato!

Approccio consigliato: meccanismo basato sulle eccezioni *dove divido il codice funzionale (TRY) da quello che tratta gli errori + CATCH (eccezioni)*

- ***Errors:*** sottoclasse di Throwable che si riferisce a gravi errori all'interno del sistema: Errore di linking, errore di virtual machine
- ***Exception:*** 2 tipologie:

A) Checked: obbligati a gestore gli errori contenendoli in blocchi tra catch
B) Unchecked: non sono controllate, sono troppo comuni e sporcherebbero il codice

+ lungo la gerarchia dei metodi dalla funzione che genera errori al main si può intercettare la esecuzione e gestirla —> metodi possono fare 3 cose:

A) **Senza delega:** intercetto l'eccezione
B) **Delega completa:** annoto il metodo delegante con **THROWS** e nome dell'eccezione delegando il chiamante —> ignoro eccezione delegandola al chiamante
C) **Delega parziale:** intercetto l'eccezione e ne produco una nuova tramite la parola **THROW**