Programmazione II – Java

Ogni file contiene una classe → nome classe = nome file

Definire una classe → campi (proprietà), costruttore (non obbligatorio) e metodi

Variabili

- tipi primitivi (iniziale minuscola) → *contengono il valore:*
  `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`
  variabili (C) → proprietà di una classe (Java)
- oggetti → *contengono il puntatore:*
  `String`, ..., o altri oggetti creati da altre classi tramite il costruttore della classe,
  comprende il costruttore e i relativi metodi di quell'oggetto,
  se un metodo è `static` si chiama tramite il nome della classe (*nome_classe.nome_metodo_static*)

Una variabile di tipo non primitivo contiene il puntatore all'oggetto.

I metodi che lavorano con oggetti ritornano il puntatore, quindi fanno l'operazione richiesta passando per il puntatore, quindi se si hanno più Collection con gli stessi elementi, possono ritrovarsi modificati in tutte le loro occorrenze.

`final` (costante) → quando viene assegnato un valore a una proprietà, questo non cambia più,
se il metodo è final, nelle classi ereditate non si può sovrascrivere.

`static` (unico) → alla compilazione del codice viene istanziato una sola volta
Se si creano più oggetti a partire dallo stesso "stampino" (classe), se un metodo (o una proprietà) è static, viene creato una vota sola, e se si modifica, si modifica per tutti gli oggetti che lo comprendono.
Può essere chiamato solo dalla classe o da un altro metodo static presente nella classe.

Visibilità

`public` → visibile ovunque

`private` → visibile solo all'interno della classe, (incapsulamento)
classi esterne non possono modificare direttamente il valore, si passa attraverso metodi pubblici, quindi anche attraverso controlli all'interno della classe:
getter → per vederlo all'esterno della classe
setter → per modificarlo

`protected` → visibile solo all'interno della cartella stessa (neanche in sottocartelle)

`package` → percorso di dove si trova il file della classe rispetto a `src` (cartella source che contiene il codice)

`this.` → sostituto dell'oggetto dentro sé stesso (segnaposto dell'oggetto stesso)

Stampa su terminale

`System.out.println("Hello world");` il println chiama già il *.toString()* dell'oggetto "più vicino"

```
import java.lang.System;
```
java.lang già importata di default

`err` → costante della classe che fa riferimento allo standard error
`in` → costante della classe che fa riferimento allo standard input
`out` → costante della classe che fa riferimento allo standard output
`static long currentTimeMillis()` → ritorna il numero di millisecondi passati dalla mezzanotte dell'1 gennaio 1970 UTC

```
System.out.println(card2);
```

```
System.out.println("ciao");
```

```
System.out.println(new Formation433(players).isValid());
System.out.println(new Formation433(players));
```

```
catch (IllegalArgumentException e) {
    System.out.println(e);
}
```

```
System.out.println(Arrays.toString(array));
```

```
System.out.println(new DecimalNumber(n) + "\n" + new BinaryNumber(n) + "\n" +
    new BinaryNumberWithParity(n) + "\n" + new OctalNumber(n) + "\n" +
    new HexNumber(n) + "\n" + new Base58Number(n));
```

Lettura dell'input da tastiera

```
import java.util.Scanner;
```

`Scanner(source)` → costruttore, che crea uno Scanner legato alla sorgente indicata
`void close()` → chiude lo Scanner: dopo non può più essere usato
`double nextDouble()`
`float nextFloat()`
`int nextInt()`
`String nextLine()`
`long nextLong()`

```
Scanner scanner = new Scanner(System.in);
int numero = scanner.nextInt();
```
Scanner(source)

caso particolare delle stringhe

Creare nuova stringa inizializzata:
`String string = new String("ciao");` → `String string = "ciao";`
Creare nuova stringa vuota:
`String string = new String();` → `String string = "";`

Identità delle stringhe (`==`) → puntano allo stesso oggetto
Uguaglianza delle stringhe (`equals()`) → controlla se un oggetto è uguale a un altro

Concatenazione fra stringhe → `+` (uso implicito dei metodi concat() e valueOf() di java.lang.String)

```
import java.lang.String;
```
java.lang già importata di default
classe immutabile → i suoi oggetti non possono più essere modificati dopo la creazione

`String(String other)` → costruttore di copia: crea un clone
`char charAt(int index)` → ritorna il char all'indice passatogli
`int compareTo(String other)` → ritorna 1 (this > other), 0 (this = other), -1 (this < other) per ordinare
`int compareToIgnoreCase(String other)` → ritorna negativo, zero, oppure positivo
ignora maiuscole/minuscole
`String concat(String other)` → implicitamente usato per la concatenazione con +
`boolean endsWith(String end)`
`boolean equals(Object other)` → controlla se 2 oggetti stringa sono uguali
(non se puntano allo stesso spazio in memoria → == con gli oggetti)
`boolean equalsIgnoreCase(String other)`
`static String format(String format, Object... args)`
`int indexOf(int character)` → posizione della prima cella che contiene il carattere passato (?)
`int indexOf(String what)` → posizione della cella dove inizia la sottostringa (?)
`boolean isEmpty()` → stringa vuota
`int length()` → lunghezza stringa (dimensione vera, non da 0)
`boolean startsWith(String what)`
`String substring(int start)` → da start incluso, estrae una stringa
`String substring(int start, int end)` → da start incluso ad end escluso
`String toLowerCase()` → rende la stringa tutta minuscola
`String toUpperCase()` → rende la stringa tutta maiuscola
`String trim()` → rimuove caratteri vuoti da inizio e fine stringa (`"\n","\t",' '`, ...)
`static String valueOf(int i)` → esegue una conversione esplicita di tipo;
esiste per tutti i tipi primitivi, non solo per int;
implicitamente usato per la concatenazione con +
toString() (?)

```
@Override
protected char getCharForDigit(int digit)  {
    return string.charAt(digit);      //ritorna il char (della stringa string)
                                      che corrisponde all'indice in input (digit)
}

private static final String string = "0123456789";
```

```
@Override
protected char getCharForDigit(int digit) { return string.charAt(digit); }

private static final String string =
    "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz";
```

numeri Random

```
import java.util.Random;
```

```
boolean nextBoolean()
double nextDouble()
float nextFloat()
int nextInt()
int nextInt(int bound) → restituisce un numero casuale tra 0 e bound escluso
long nextLong()
```

```
Random random = new Random();
valore = random.nextInt();  //genera un numero random
valore = random.nextInt(bound: 13);  //genera un numero random tra 0 e bound escluso
```

Operazioni aritmetiche

- Per operazioni tra i tipi primitivi → `+`, `-`, `*`, `/`, `==`, `!=` ...
- Per operazioni che comprendono oggetti → si passa per un metodo (es. `this.value.equals(other.value)`)
  `equals()` → controlla se un oggetto (`this`) è uguale a un altro oggetto (`other`)
  `==` → controlla se 2 oggetti puntano alla stessa cella di memoria

La libreria Math non ha costruttore, per chiamare un metodo che esegue un'operazione → `Math.`*nome_metodo*

```
import java.lang.Math;
```

```
static double E → costante e
static double PI → costante π
static int abs(int i) → modulo, esiste anche per altri tipi numerici
static double cos(double d) → coseno
static double log(double d) → logaritmo in base e
static double log10(double d) → logaritmo in base 10
static int max(int a, int b) → esiste anche per altri tipi numerici
static int min(int a, int b) → esiste anche per altri tipi numerici
static double sin(double d) → seno
static double sqrt(double d) → radice quadrata
static double tan(double d) → tangente
static double toDegrees(double radiants)
static double toRadiants(double degrees)
```

Istruzioni if, while, do, for e switch

Quasi identiche a quelle del linguaggio C.

for each

```
int portieri = 0;

/*
for (int i = 0; i < players.length; i++) {      //for normale
    if (players[i].canUseHands()) {
        portieri++;
    }
    if (portieri > 1) {
        return false;
    }
}
*/

for (SoccerPlayer player : players) {       //for each → SoccerPlayer player = players[i];
                                            //viene istanziata una variabile, in questo caso di tipo SoccerPlayer
                                              (bisogna prendere il tipo della proprietà della singola cella
                                              dell'array → ad esempio con String si prende char
                                              (ma il for each non funziona con le stringhe)),
                                              che passa una a una ogni cella dell'array
    if (player.canUseHands()) {
        portieri++;
    }
    if (portieri > 1) {
        return false;
    }
}

return players.length == 11 && portieri == 1;
```

```
@Override
public String toString(){
    String string = super.toString();
    int counter1 = 0;

    for (int i = 0; i < string.length(); i++) {
        //char c = string.charAt(i);
        if (string.charAt(i) == '1') {
            counter1++;
        }
    }
    /*
    for (char stringa : string) {     //non si può fare il for each sulle stringhe
        if (stringa == 1) {
            counter1++;
        }
    }
    */

    return string + (counter1 % 2);  //se dispari il resto è 1, senno è 0
                                     //viene concatenato alla stringa del numero binario già esistente
}
```

Mettere esempi

Length

For each

Inizializzazione for normale

Altro

Ritorna puntatore

```
int[] array = new int[10];        //array vuoto che conterrà tipi primitivi int
                                  (già inizializzato tutto a 0)
```

```
String[] array = new String[10];      //array vuoto che conterrà il puntatore ad obj di tipo String
```

```
Number[] array = {new DecimalNumber(2024), new BinaryNumber(113),
    new BinaryNumberWithParity(158), new OctalNumber(827), new HexNumber(2066),
    new Base58Number(8092)};
```

```
private static final char[] array =
    {'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
```

```
String[] difensori = {"Alex Sandro", "Rugani", "Chiellini", "Dani Alves"};
String[] centrocampisti = {"Fabinho", "Iniesta", "Pjanic"};
String[] attaccanti = {"Dybala", "Higuain", "Bernardeschi"};
String[] portiere = {"Szczesny"};
String[][] giocatori = {difensori, centrocampisti, attaccanti, portiere}; //matrice

SoccerPlayer[] players = new SoccerPlayer[11];   //array di SoccerPlayer con dimensione definita
```

```
for (int i = (elements.length - 1); i >= 0 ; i--) //cicla tutti gli elementi dell'array
                                                   (fino a i < elements.length)
                                                   (elements.length → lunghezza array)
```

```
import java.util.Arrays;
```

`static int binarySearch(int[] arr, int key)` → ritorna la posizione di *key* dentro *arr*, ritorna un numero negativo se *arr* non contiene *key*.
Assume che l'array arr sia ordinato.
Questo metodo esiste anche per gli altri tipi primitivi numerici e tipi riferimento (contiene puntatore agli oggetti al suo interno), (chiama *compareTo()* per decidere l'ordine).

`static boolean equals(int[] arr1, int[] arr2)` → controlla che *arr1* e *arr2* abbiano stessa lunghezza e contengano gli stessi elementi nello stesso ordine.
Questo metodo esiste anche per gli altri tipi primitivi e tipi riferimento (contiene puntatore agli oggetti al suo interno), (chiama *equals()* fra tutte le coppie di oggetti da confrontare).

`static void fill(int[] arr, int val)` → assegna *val* a tutti gli elementi di *arr*.
Questo metodo esiste anche per tutti gli altri tipi primitivi e tipi riferimento (contiene puntatore agli obj al suo interno)

`static void sort(int[] arr)` → ordina *arr* in senso crescente.
Questo metodo esiste anche per tutti gli altri tipi primitivi numerici e tipi riferimento (contiene puntatore agli oggetti al suo interno), (chiama *compareTo()* per decidere l'ordine).

`static String toString(int[] arr)` → ritorna una stringa che riporta gli elementi di arr, nel loro ordine [0, 1, 2, 3, ..., n-1]
Questo metodo esiste anche per gli altri tipi primitivi e tipi riferimento (contiene puntatore agli oggetti al suo interno), (chiama *toString()* di *Arrays* sugli elementi dell'array e concatena il risultato).

```
Arrays.sort(array); //ordina l'array, secondo il metodo compareTo (ordinamento crescente) → Comparable<T>
System.out.println(Arrays.toString(array)); // [1110001, 100111101, 1473, 2024, 812, 3QX]
                                            //qui non è ridondante
                                              perché bisogna chiamare il toString degli array
```

Costruttore

Ha lo stesso nome della classe, assegna i valori alle proprietà,
non ha *return*
Può essercene uno o più, si differenziano dal numero e dal tipo di parametri che ricevono in input

Una classe può anche non avere un costruttore, ad esempio la classe *Math*, che h solo costanti e metodi statici, oppure averne uno che però completamente vuoto, sia negli input che nel corpo

```
public class Card {

    private final int value;
    private final int suit;

    // Genera una carta a caso con un valore da min (incluso) in su.
    // Param: min - il valore minimo (0-12) della carta che può essere generata
    public Card(int min) {      //costruttore
        Random random = new Random();
        int valore;
        do {
            valore = random.nextInt(bound: 13);  //genera un numero random tra 0 e bound escluso
        } while (valore < min);

        value = valore;
        // value = random.nextInt(13 - min) + min;

        suit = random.nextInt(bound: 4);
    }

    // Genera una carta a caso con un valore da 0 (incluso) in su.
    public Card() {       //costruttore vuoto (non ha parametri in input)
        // this(0);       //si può richiamare il primo costruttore passando il parametro più generico

        Random random = new Random();
        value = random.nextInt(bound: 13);

        suit = random.nextInt(bound: 4);
    }
```

Utilizzo

```
String card1 = new Card()     //crea un nuovo oggetto di tipo Card
String card1 = (new Card()).toString();      //crea un nuovo oggetto di tipo Card e ne fa il toString()

String card2;
do {
    card2 = new Card().toString();
    System.out.println(card2);
} while (!card1.equals(card2));
```

Utilizzo

```
Card card1 = new Card(min: 12);
System.out.println(card1);
System.out.println(card1.toString());        //il toString() nel println è ridondante
                                              perché chiama già il toString della classe più vicina

Card card2 = null;
do {
    card2 = new Card(min: 12);
    System.out.println(card2);
} while (!card1.equals(card2));
```

```
Number[] array = {new DecimalNumber(2024), new BinaryNumber(113),
    new BinaryNumberWithParity(158), new OctalNumber(827), new HexNumber(2066),
    new Base58Number(8092)};
```

parametri Varargs

Passati in come ultimo input un numero variabile di parametri, poi trasformati in array

Vincoli sulle variabili di tipo ???

```
public List(T head, T... elements) {   //passa in input in numero indefinito di elements di tipo T
                                        //siccome non si sa quanti sono,
                                          sono gli ultimi parametri passati (possono essere 0 come ∞)
    this.head = head;

    List<T> list = null;
    for (int i = (elements.length - 1); i >= 0 ; i--) //cicla tutti gli elementi dell'array
                                                        (fino a i ≺ elements.length)
                                                        (elements.length → lunghezza array)
        list = new List<T>(elements[i], list);
```

```
private final T head;    //testa della lista
private final List<T> tail;    //il resto della lista

// crea una lista con la testa e la coda indicate
public List(T head, List<T> tail) {
    this.head = head;    //testa
    this.tail = tail;    //elemento successivo (next)
}

// crea una lista contenente la testa indicata, seguita dagli elementi indicati
public List(T head, T... elements) {   //passa in input in numero indefinito di elements di tipo T
    this.head = head;    //assegnata la testa ("value")

    /*
    T[] elementi = new T[elements.length];   //non si può istanziare un array di tipo T
    for (int i = 0; i < elements.length-1; i++) {
        elementi[i] = elements[i+1];
    }
    this.tail = new List<T>(elements[0], elementi);
    */

    List<T> list = null;

    for (int i = (elements.length - 1); i >= 0 ; i--) {   //metodo ricorsivo
                                                          //abbiamo già la testa (this.head = head;)
                                                        //parte dall'ultimo elemento della lista, costruendo ogni nodo
                                                          tramite il primo costruttore, e salvata mano a mano in list
        list = new List<T>(elements[i], list);    //prende in input "testa" e "elemento successivo"
    }

    this.tail = list;    //alla fine "collega" la testa alla coda (la coda è l'elemento successivo ("next"))
}
```

Utilizzo

```
IntList l2 = new IntList(head: 11, ...elements: 13, 42, 9, -5, 17, 13);
```

Classe di costanti

nome_classe_enumerazione.
`Static E[] values()` → ritorna l'array di tutti gli elementi dell'enumerazione `static E valueOf(String name)` → ritorna l'elemento dell'enumerazione che ha il nome indicato `int compareTo(E other)` → *compareTo()* sugli elementi dell'enumerazione `int ordinal()` → ritorna il numero d'ordine (indice) di un elemento dell'enumerazione

Value.java (enumerazione)

```
public enum Value {
    DUE,
    TRE,
    QUATTRO,
    CINQUE,
    SEI,
    SETTE,
    OTTO,
    NOVE,
    DIECI,
    J,
    Q,
    K,
    ASSO;
}
```

Suit.java (enumerazione)

```
public enum Suit {
    PICCHE,
    FIORI,
    QUADRI,
    CUORI;
}
```

Utilizzo in una classe

```
public Card2(int min) {
    Random random = new Random();
    int valore;
    do {
    valore = random.nextInt(bound: 13);
    } while (valore < min.ordinal());        //ordinal() ritorna l'indice della posizione del "segnaposto" all'interno dell'enumerazione

    value = (Value.values())[valore];    // values() ritorna a modi array i valori dentro Enum
    // value = random.nextInt(13 - min) + min;

    suit = Suit.values()[random.nextInt(bound: 4)];
}

// Genera una carta a caso con un valore da 0 (incluso) in su.
Public Card2() { this(Value.DUE); }

// Ritorna una descrizione della carta sotto forma di stringa, del tipo 10♠ oppure J♥.
Public String toString() {
    String[] valori = {"2","3","4","5","6","7","8","9","10","J","Q","K","1"};
    String[] semi = {"♠","♣","♦","♥"};

    return valori[value.ordinal()] + semi[suit.ordinal()];
}
```

Utilizzo nel main

```
Card2 card1 = new Card2(Value.DUE);
System.out.println(card1);
Card2 card2;
do {
    card2 = new Card2(Value.DUE);
    System.out.println(card2);
} while (!card1.equals(card2));
```

Color.java *(enumerazione)*

```
package it.univr.figures;

public enum Color {
    GIALLO,
    ROSSO,
    BLU,
    VERDE,
    NERO;
}
```

```
public GreenDot() { super(Color.VERDE, 1); }
```

```
public static void main(String[] args) {
    Figure rettangolo = new Rectangle(Color.BLU, base: 10, altezza: 11);
    Figure quadrato = new Square(Color.BLU, lato: 10);
    print(rettangolo);

    Figure cerchio = new Circle(Color.GIALLO, raggio: 5);
    print(cerchio);

    //in questa classe non è possibile chiamare il metodo getColor() sulle figure,
      perché il metodo è protected e la classe main non si trova nella stessa cartella del file che contiene il metodo
      getColor() da richiamare
}
```

Interfacce

Qui vengono specificati i metodi da implementare nella sua sottoclasse.
La visibilità delle interfacce e dei loro metodi è implicitamente public.

SoccerPlayer.java *(interfaccia)* comprende tutti i metodi di una classe, senza implementarli

```java
Specifica un giocatore di calcio
public interface SoccerPlayer {
    String toString();  // ritorna il nome del giocatore
    boolean canUseHands();  // determina se il giocatore può usare le mani
}
```

Number.java *(interfaccia)*

```java
public interface Number extends Comparable<Number> {   //Comparable<T>
    int getValue(); // restituisce il valore di questo numero
}
```

Classi Astratte

Ibrido tra classe concreta e interfaccia,
contiene sia metodi già implementati, sia metodi da implementare nelle sottoclassi, contrassegnati con abstract

AbstractSoccerPlayer.java *(classe astratta)* mette insieme la classe concreta e l'*interfaccia*, comprende quindi metodi già implementati, e metodi non implementati (da implementare nelle sottoclassi), contrassegnati dalla parola `abstract`

```java
public abstract class AbstractSoccerPlayer implements SoccerPlayer { //implements → implementa l'interfaccia

    private String name;

    protected AbstractSoccerPlayer(String name) { this.name = name; }

    @Override
    public final String toString() { return name; }

    public abstract boolean canUseHands(); //metodo da implementare nella sottoclasse
}
```

```
nome_sottoclasse extends nome_superclasse
```

Si può creare una classe figlia (*extends*) (sottoclasse) che eredita quindi i metodi della classe madre (superclasse).

super. super()

Per richiamare i metodi della classe madre dalla classe figlia si usa *super.metofo_da_richiamare*.
Se la superclasse ha un costruttore, per passare in input i parametri al costruttore della superclasse si utilizza *super(parametri_da_passare)* (nella prima riga del costruttore),
se non lo si fa è implicito *super()*, che chiama il costruttore vuoto.

Ogni classe ha cmq i suoi costruttori e questi non vengono ereditati dalle sottoclassi.
Le proprietà, anche se assegnate da una sottoclasse, rimangono proprie della loro classe di appartenenza, quindi, ad esempio se una proprietà è private nella superclasse, non si può vedere nella sottoclasse, tranne se non con un getter.

@Override

La classe eredita i metodi public (o protected se nella stessa cartella), i metodi private ci sono ma non si vedono (che se pur visibili non possono essere usati al di fuori della classe in cui di trovano.
Si può cmq sovrascrivere (@Override) un metodo ereditato dalla superclasse,
ma per far sì che sia quel metodo a essere sovrascritto, quindi quello a essere richiamato in caso venga chiamato il metodo per la sottoclasse, deve avere la stessa firma (tipo di ritorno – nome metodo – lista di input (quantità, tipo e ordine)).
Infatti quando il metodo viene chiamato, viene chiamato il metodo della classe più vicina.

La sottoclasse può implementare suoi metodi.

cosa estendente, quali metodi usa, tipi passati in input

Java può estendere una sola classe, ma implementare più interfacce.

Un metodo può ricevere in input il tipo di un'interfaccia o della superclasse,
in questo modo accetta l'interfaccia o la superclasse stessa,
ma anche la sottoclasse ma con il tipo dell'interfaccia o della classe stessa
(ma utilizza l'implementazione dei metodi della sottoclasse (classe più vicina), però può vedere solo metodi definiti dalla superclasse passata in input, perché di tipo della superclasse)

casting

Per ovviare si usa il casting dopo aver controllato (instanceof)

```
tipo_passato_in_input nome = (tipo_variabile_castata) nuovo_nome;
```

interfaccia → implements
classi → extends

Rectangle di *Figure*, rappresenta un rettangolo.

```
public class Rectangle extends Figure{      //extends → Rectangle è figlia di Figure
    private double base;
    private double altezza;

    public Rectangle(Color colore, double base, double altezza) {
        super(colore);   //richiama il costruttore della superclasse
                         e assegna il valore alla rispettiva proprietà, già presente nella superclasse
                         (quindi non si deve ri-istanziare nella sottoclasse).
                         //va richiamato nella prima riga di codice del costruttore,
                         se non si scrive, è implicito super();
        this.base = base;
        this.altezza = altezza;
    }

    @Override
    public double perimeter() { return (base + altezza)*2; }

    @Override
    public double area() { return base*altezza; }

    @Override
    Ridefinito per ritornare la stringa "Rectangle of " seguita dalla chiamata al toString() della superclasse.
    public String toString() { return "Rectengle of " + super.toString(); }
}
```

Square di *Rectangle*, rappresenta un quadrato. I metodi *double perimeter()* e *double area()* non vengono ridefiniti.

```
public class Square extends Rectangle{

    public Square(Color colore, double lato) { super(colore, lato, lato); }

    @Override
    public String toString() { return "Square, a " + super.toString(); }
}
```

```
public static void main(String[] args) {
    Figure rettangolo = new Rectangle(Color.BLU, base: 10, altezza: 11);
    Figure quadrato = new Square(Color.BLU, lato: 10);
    print(rettangolo);

    Figure cerchio = new Circle(Color.GIALLO, raggio: 5);
    print(cerchio);

    //in questa classe non è possibile chiamare il metodo getColor() sulle figure,
      perché il metodo è protected e la classe main non si trova nella stessa cartella del file che contiene il metodo
      getColor() da richiamare
}
```

```
public static void print(Figure figure) {   //l'oggetto più generico può contenere gli oggetti più
                                              specifici, quindi posso passare Rectangle,
                                              che è una sottoclasse di Figure,
                                            //ma viene poi utilizzata sotto il tipo di Figure,
    if (figure instanceof Rectangle)   //per averla sotto il tipo di Rectangle, dopo aver controllato
                                       che effettivamente è un'istanza di Rectangle,
                                       si può castare, ovvero creare una nuova variabile,
                                       definendola col tipo più specifico,
                                       oppure si può dire di considerarla del tipo più specifico,
                                       (si deve fare ogni volta che si utilizza)
```

```java
@Override
    public final boolean equals(Object other) {
        if (!(other instanceof Number)) {  //controlla se other è istanza di Number,
                                           se non lo è si sa già che non è uguale
            return false;
        }

        //other = (Number) other;     NO → ridondante perché other resta di tipo Object
        //((Number) other).getValue();   scrivere ogni volta il tipo da considerare

        Number otherNumb = (Number) other; //cast
        return this.value == otherNumb.getValue();
    }
```

Sottoclassi *Forward*, *Midfield*, *Defence* e *GoalKeeper*, (solo il *GoalKeeper* può usare le mani). ...

Forward.java (classe concreta)

```java
public class Forward extends AbstractSoccerPlayer{

    protected Forward(String name) { super(name); }     //anche il costruttore della
                                                        superclasse è protected,
                                                        cambiando visibilità non si va a
                                                        sovrascrivere, ma se ne crea uno
                                                        nuovo
    @Override
    public boolean canUseHands() { return false; }
}
```

```java
if (!super.isValid()) { return false; }   // la superclasse fa già parte del controllo

int difensori = 0;
int centrocampisti = 0;
int attaccanti = 0;

for (SoccerPlayer player : getPlayers()) {
    if (player instanceof Defence) {
        difensori++;
    } else if (player instanceof Midfield) {
        centrocampisti++;
    } else if (player instanceof Forward) {
        attaccanti++;
    }
}

return difensori == 4 && centrocampisti == 3 && attaccanti == 3;
```

```java
public static void test (List<SoccerPlayer> playerList) {}   //LinkedList
                                    //riesce a entrare perché la classe LinkedList è figlia dell'interfaccia List
                                    //e il metodo è static come il main (chiamato dal main)
```

toString()

```
// Ritorna una descrizione della carta sotto forma di stringa, del tipo 10♠ oppure J♥.
public String toString() {
    String[] valori = {"2","3","4","5","6","7","8","9","10","J","Q","K","1"};
    String[] semi = {"♠","♣","♦","♥"};

    return valori[value] + semi[suit];
}
```

```
@Override     //indica che si sta andando a sovrascrivere il metodo di una classe superiore
    public String toString() {
        return "area: " + this.area() + ", perimeter: " + this.perimeter() +
                ", color: " + colore;
    }
```

implementati per l'ordinamento
o un ordinamento o l'inserimento ordinato di una struttura dati.
Nel caso dell'*equals*, utilizzato anche ad esempio per verificare se un oggetto è contenuto in una struttura dati, o per altre operazioni che richiedono l'*equals*

equals() → fare *@Override* dalla classe *Object* (riscrivere *equals* → *equals* di *obj* (==))
di solito riscritto, perché quello ereditato della classe *Object* utilizza ==,
che nel caso degli oggetti non ne verifica l'effettiva uguaglianza, ma se puntano alla stessa cella di memoria

compareTo → implementato della classe *Comparable<T>*

`import java.lang.Comparable<T>;` interfaccia
`int compareTo(T other)` → ritorna un numero negativo se *this* > *other*, un numero positivo se viene prima *this* < *other*, ritorna 0 se *this* e *other* sono equivalenti (*this.equals(other)* per obj o *this* == *other* per tipi primitivi)

```java
// Determina se questa carta è uguale ad other.
    // Param: other - l'altra carta con cui confrontarsi
    // Return: true se e solo se le due carte sono uguali
    public boolean equals(Card other) {
        return value == other.value && this.suit == other.suit;     //ritorna il risultato della condizione
    }

    // Ordina le carte, prima in base al valore, poi in base al seme.
    // Param: other - l'altra carta con cui confrontarsi
    // Return: -1 se this < other, 0 se sono uguali, 1 se this > other.
    public int compareTo(Card other) {
        if (this.value!= other.value) {
            if (this.value > other.value) {
                return 1;
            } else {
                return -1;
            }
        } else {
            if (this.suit != other.suit) {
                if (this.suit > other.suit) {
                    return 1;
                } else {
                    return -1;
                }
            } else {
                return 0;
            }
        }
    }
```

Utilizzo *equals()*

```java
String card2;
do {
    card2 = new Card().toString();
    System.out.println(card2);
} while (!card1.equals(card2));
```

```java
// due numeri sono uguali se e solo se hanno lo stesso valore
@Override
public final boolean equals(Object other) {
    if (!(other instanceof Number)) {       //controlla se other è istanza di Number,
        return false;    //se non lo è, si sa già che non è uguale
    }
    return this.value == ((Number) other).getValue();
}

// ordinamento fra i Number è quello crescente per valore
@Override
public final int compareTo(Number other) {
    if (this.value < other.getValue()) {
        return -1; //ritorna -1 se this < other
    } else if (this.value == other.getValue()) {
        return 0;  //ritorna 0 se this = other
    } else {
        return 1;  //ritorna 1 se this > other
    }
}
```

Classi wrapper della libreria standard

- Integer
- Character
- ...

Boxing e unboxing automatico da tipi primitivi a classi wrapper e viceversa tramite assegnamento (?)

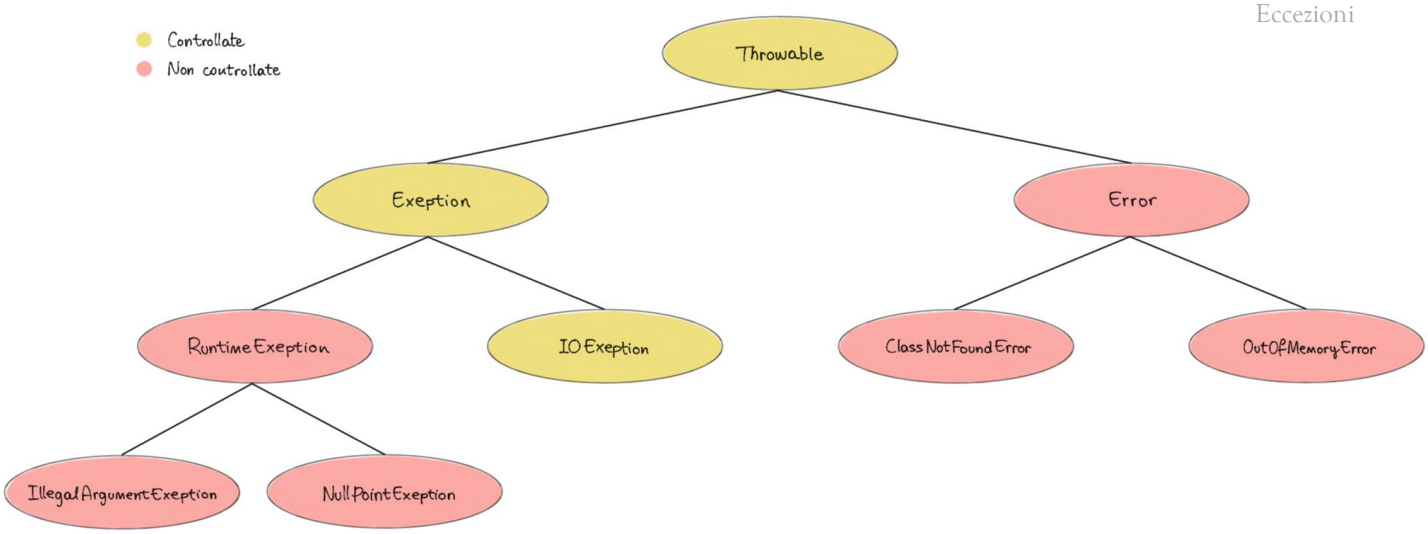
```
import java.lang.Integer;
```

`static int MAX_VALUE` → costante max int
`static int MIN_VALUE` → costante min int
`Integer(int value)` → deprecato!
`Integer(String value) throws java.lang.NumberFormatException`
`int intValue()` → restituisce il valore int corrispondente *(per convertire a tipo primitivo int)*
`int compareTo(Integer other)` → Integer implementa Comparable<Integer>
`static int parseInt(String s) throws java.lang.NumberFormatException` → da String a int
`static String toBinaryString(int i)` → ritorna la rappresentazione binaria di i
`static String toHexString(int i)` → ritorna la rappresentazione esadecimale di i
`static Integer valueOf(int i)` → ritorna new Integer(i) ma usa una cache per chiamate ripetute *(per convertire a oggetto Integer)*

classi wrapper simili corrispondenti agli altri tipi primitivi:
`java.lang.Short`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`, `java.lang.Byte` e `java.lang.Boolean`.

```
import java.lang.Character;
```

`static char MAX_VALUE` → costante max char
`static char MIN_VALUE` → costante min char
`Character(char value)` → deprecato!
`char charValue()` → restituisce il valore char corrispondente *(per convertire a tipo primitivo char)*
`int compareTo(Character other)` → infatti Character implementa Comparable<Character>
`static boolean isDigit(char c)`
`static boolean isLetter(char c)`
`static boolean isLetterOrDigit(char c)`
`static boolean isLowerCase(char c)`
`static boolean isUpperCase(char c)`
`static boolean isWhitespace(char c)`
`static char toLowerCase(char c)`
`static char toUpperCase(char c)`
`static Character valueOf(char c)` → ritorna new Character(c) ma usa una cache per chiamate ripetute *(per convertire a oggetto Character)*



throw new eccezione()

throws per costruttori e metodi

try/catch/finally

definizione di nostre classi di eccezione

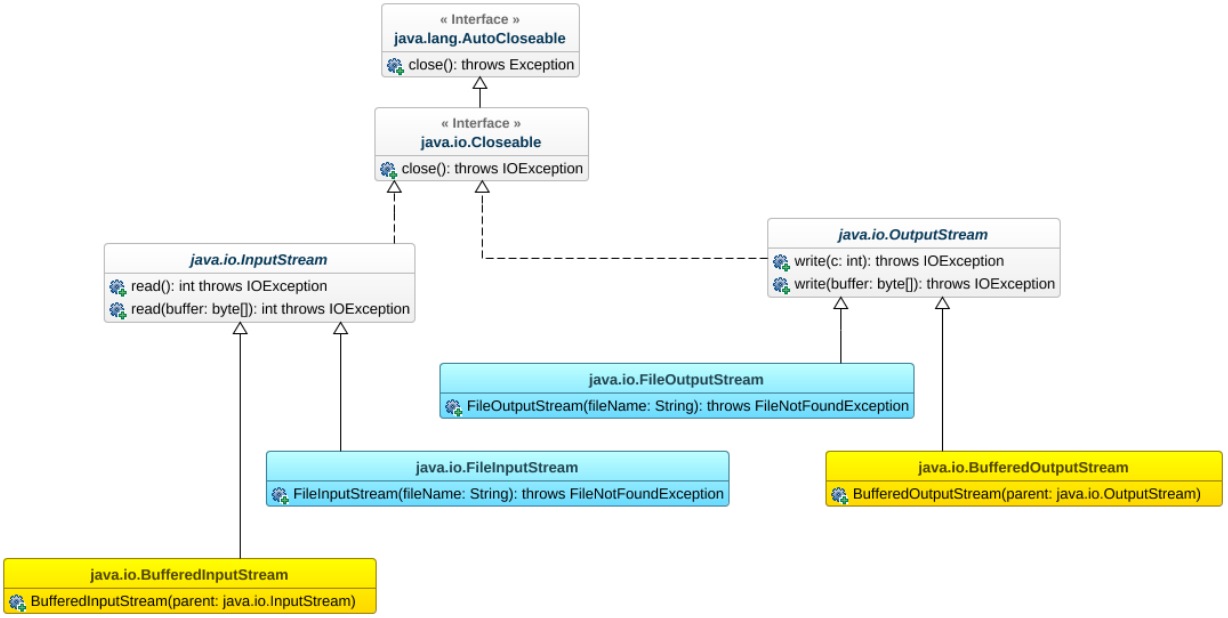
costruttore senza input

costruttore con input

```
if (!isValid())
    throw new IllegalArgumentException("invalid formation");     //lancia un errore (rosso)
```

```
// se value è negativo, esegue throw new IllegalArgumentException(); altrimenti inizializza il campo value
    protected AbstractNumber(int value) {
        if (value < 0) {
            throw new IllegalArgumentException();
        }
        this.value = value;
    }
```

```java
//try – catch – finally
try {       //prova il pezzo di codice all'interno,
            se durante l'esecuzione vengono generate delle eccezioni all'interno del try,
            e sono catturabili da uno dei catch, esegui il catch corrispondente all'eccezione (errore)
    System.out.println(new Formation433(players).isValid());
    System.out.println(new Formation433(players));
} catch (ExceptionInInitializerError e) {  //esegue in catch sse l'eccezione (errore) generata
                                                è del tipo che riceve come parametro o sua "figlia"
    System.out.println("Errore nella creazione della formazione.");
    //System.exit(-1);          //-1 → codice di uscita con errore
} catch (IllegalArgumentException e) {
    System.out.println(e);
    //System.exit(-1);
} finally { //a prescindere esegui,
                se non viene terminata prima l'esecuzione (System.exit, errori non catturati, return, ...)
    System.out.println("Try catch finito.");
    System.exit(-1);
}
```



Leggere file di caratteri

```
import java.io.Reader;
```
classe astratta

```
int read() throws java.io.IOException
```
→ blocca l'esecuzione finché non c'è un carattere da leggere;
a quel punto ritorna il codice Unicode del prossimo carattere letto;
ritorna -1 se la sorgente di lettura è terminata

```
int read(char[] buffer) throws java.io.IOException
```
→ blocca l'esecuzione finché non arriva qualche carattere da leggere;
a quel punto scrive i caratteri nel buffer e ritorna il numero di caratteri letti;
ritorna -1 se la sorgente di lettura è terminata

```
import java.io.FileReader;
```

- metodi ereditati da `java.io.Reader`
- metodi

```
FileReader(String fileName) throws java.io.FileNotFoundException
```
→ crea un lettore di file che legge i caratteri dal file di testo col nome indicato

```
import java.io.BufferedReader;
```

- metodi ereditati da `java.io.Reader`
- metodi

```
BufferedReader(Reader parent)
```
→ crea una vista bufferizzata di *parent*

Scrivere file di caratteri

```
import java.io.Writer;    classe astratta
```

```
void write(int c) throws java.io.IOException → interpreta i 16 bit meno significativi di c
                                                come codice Unicode di un carattere
                                                e lo scrive nel file
void write(char[] buffer) throws java.io.IOException → scrive nel file i caratteri contenuti
                                                       nell'array buffer
void write(String s) throws java.io.IOException → scrive nel file i caratteri della stringa s
```

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

```
- metodi ereditati da java.io.Writer
- metodi
FileWriter(String fileName) throws java.io.IOException → crea uno scrittore di file
                                                         che scrive i caratteri nel file di testo
                                                         col nome indicato
```

```
import java.io.BufferedWriter;
```

```
- metodi oltre quelli ereditati da java.io.Writer
- metodi
BufferedWriter(Writer parent) → crea una vista bufferizzata di parent
```

scrivere comodamente file di caratteri

```
import java.io.PrintWriter;
```

```
- metodi oltre quelli ereditati da java.io.Writer
- metodi
PrintWriter(String fileName) throws java.io.FileNotFoundException → crea uno scrittore
                                                                    di file
                                                                    che scrive i
                                                                    caratteri nel file di
                                                                    testo col nome
                                                                    indicato
void print(int i) → scrive i caratteri della rappresentazione decimale dell'intero i nel file.
                    Questo metodo esiste anche per gli altri tipi primitivi
void println(int i) → scrive i caratteri della rappresentazione decimale dell'intero i nel file.
                      Questo metodo esiste anche per gli altri tipi primitivi
void print(String s)
void println(String s)
PrintWriter printf(String format, Object... args) → scrive il formato nel file,
                                                    in stile linguaggio C
```

```
import java.lang.AutoCloseable;
```

```
void close() throws java.lang.Exception
```

```
import java.io.Closeable;
```

```
void close() throws java.io.IOException
```

List.java (classe concreta)

```
// scrive gli elementi di questa lista (cioè il loro toString())
// dentro il file testuale col nome indicato (un PrintWriter vi aiuterà)
public void dump(String fileName) throws IOException{  //dump = "buttare fuori" →
                                                        In questo caso scrivere su file
                                                        (struttura dati salvata nella ram)
    PrintWriter printWriter = new PrintWriter(fileName);   //per sola scrittura su file

    printWriter.print(this.head + " ");
    List<T> tempTail = tail;
    while (tempTail != null) { //come il toString, cicla sui nodi e li scrive mano mano
        printWriter.print(tempTail.head + " ");
        tempTail = tempTail.tail;
    }

    printWriter.close();  //chiude il file (in scrittura)
}
```

.
.
.

IntList.java (classe concreta)

```
private static IntList readFrom(Scanner scanner) throws IOException {
    try {
        //return new IntList(scanner.nextInt(), scanner.hasNextInt() ?
            readFrom(scanner) : null);      //new IntList(1, new IntList(2, new IntList(3, null)));
                                            //soluzione, ricorsiva, ma ritorna, il primo parametro (head) al
                                            costruttore, prima di fare il controllo scanner.hasNextInt(),
                                            (quindi ritorna un errore, perché non controlla il primo ma ritorna
                                            direttamente, e poi fa il controllo sui successivi),
                                            e fa il controllo sul secondo parametro (tail)
                                            //aggiungo il ritorno del primo senza controllo nel readFrom(String) *
                                            //soluzione → ritorna il primo parametro (scanner.nextInt()),
                                            poi per il secondo (tail) →
                                            if il file ha un intero successivo (scanner.hasNextInt() ?)
                                            se è true, ritorna leggi (readFrom(scanner)),
                                            sennò ( : ) ritorna null (null)
        //ricorsivo → implementata la soluzione per esteso **
        if (scanner.hasNextInt()) {  //prima di leggere,
                                            controlla se quello che andrà a leggere e un intero
            return new IntList(scanner.nextInt(), readFrom(scanner));
        } else {
            return null;
        }

    } catch (NoSuchElementException e) {   //se l'elemento letto non è un intero …
        throw new IOException(e);    //… lancia un'eccezione (di tipo IOException(e))
    }
}
```

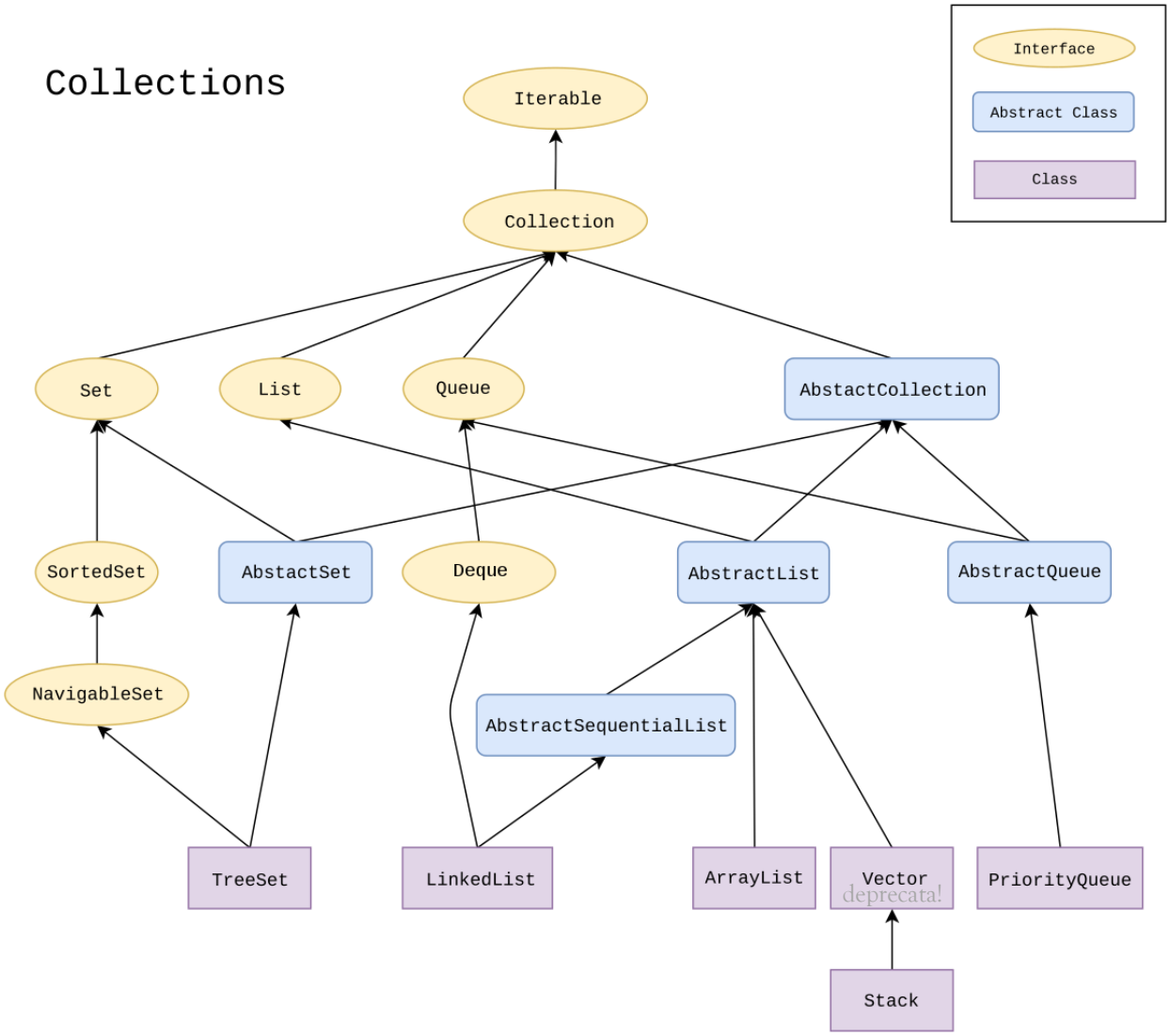
Main.java

```
try {
    List<String> l1 = new List<String>
        (head: "hello", ...elements: "how", "are", "you?");
    System.out.println(l1 + " di lunghezza " + l1.length());
    l1.dump(fileName: "l1.txt");

    IntList l2 = new IntList(head: 11, ...elements: 13, 42, 9, -5, 17, 13);
    System.out.println(l2 + " di lunghezza " + l2.length());
    l2.dump(fileName: "l2.txt");


    IntList l3 = IntList.readFrom(fileName: "l2.txt");
    System.out.println(l3 + " di lunghezza " + l3.length());

    IntList.readFrom(fileName: "l1.txt"); // fallisce perché l1.txt contiene stringhe, non interi
}
catch (IOException e) {
    //System.out.println(e);
    System.out.println("Errore di I/O");
}
```



Una variabile di tipo non primitivo contiene il puntatore all'oggetto.

I metodi che lavorano con oggetti ritornano il puntatore, quindi fanno l'operazione richiesta passando per il puntatore, quindi se si hanno più Collection con gli stessi elementi, possono ritrovarsi modificati in tutte le loro occorrenze.

`import java.util.Collection<E>;` interfaccia

`boolean add(E element)` → aggiunge un elemento, ritorna *true* se l'elemento è stato aggiunto,
per il suo *equals* (riscrivere *equals* → *equals* di *obj* (==))

`boolean addAll(Collection<E> other)` → aggiunge tutti gli elementi di una collection,
ritorna *true* se almeno un elemento è stato aggiunto,
per il suo *equals* (riscrivere *equals* → *equals* di *obj* (==))

`boolean contains(Object element)` → controlla se un elemento è contenuto nella *collection*.
per il suo equals (riscrivere *equals* → *equals* di *obj* (==))

`boolean containsAll(Collection<?> other)` → controlla se tutti gli elementi sono contenuti nella *collection*.,
se anche solo un elemento non è stato aggiunto,
ritorna *false*,
per il suo equals (riscrivere *equals* → *equals* di *obj* (==))

`boolean isEmpty()` → controlla se la lista è vuota

`boolean remove(Object element)` → rimuove dalla collection element,
ritorna *true* se l'elemento è stato rimosso,
controlla che elemento rimuovere per il suo *equals*
(riscrivere *equals* → *equals* di *obj* (==))

`boolean removeAll(Collection<?> other)` → rimuove dalla collection element,
ritorna *true* se almeno un elemento è stato rimosso,
controlla se un elemento rimuovere per il suo equals
(riscrivere *equals* → *equals* di *obj* (==))

`boolean retainAll(Collection<?> other)` → vengono passati gli elementi da mantenere, (?)
ritorna *true* se almeno un elemento è stato rimosso,
controlla se un elemento rimuovere per il suo *equals*
(riscrivere *equals* → *equals* di *obj* (==))

`int size()` → dimensione della collection

```
private SortedSet<Product> products = new TreeSet<>();

public void add(Product... products) {
    this.products.addAll(List.of(products));
}
```

`import java.util.List<E>;` interfaccia

- metodi ereditati da `java.util.Collection<E>`
- @Override metodi ereditati da `java.util.Collection<E>`

`boolean add(E element)` → aggiunge un elemento in fondo alla lista,
ritorna sempre *true* (la lista può contenere più elementi uguali)

`boolean remove(Object element)` → rimuove la prima occorrenza di element, se c'è,
ritorna *true* se l'elemento è stato rimosso,
controlla che elemento rimuovere per il suo *equals*
(riscrivere *equals* → *equals* di obj (==))

- metodi

`void add(int index, E element)` → piazza l'elemento alla posizione *index* (*i* → tra 0 e *size()* inclusi),
sposta di 1 verso destra (→) tutti gli elementi dopo *index*

`E get(int index)` → ritorna l'elemento alla posizione *index* (*i* → tra 0 e *size()* escluso)

`int indexOf(Object element)` → ritorna la prima posizione della prima occorrenza di element,
ritorna -1 se la lista non contiene element

`static <E> List<E> of(E... elements)` → *factory method*
costruisce una lista immutabile con dentro *elements*

`E remove(int index)` → rimuove e ritorna l'elemento alla posizione *index* (*i* → tra 0 e *size()* escluso),
sposta di 1 verso sinistra (←) tutti gli elementi dopo *index*

`E set(int index, E element)` → prende l'elemento alla posizione *index* (*i* → tra 0 e *size()* inclusi), ***
e lo sostituisce con element

`import java.util.LinkedList<E>;`

- metodi ereditati da `java.util.List<E>`
- metodi

`LinkedList(Collection<? extends E> parent)` → crea una lista e la riempie con gli elementi di *parent*
(*parent* è di tipo Collection o sue sottoclassi)

`import java.util.ArrayList<E>;`

- metodi ereditati da `java.util.List<E>`
- metodi

`ArrayList(Collection<? extends E> parent)` → crea una lista e la riempie con gli elementi di *parent*
(*parent* è di tipo Collection o sue sottoclassi)

```
import java.util.Set<E>;
```
interfaccia

- metodi ereditati da `java.util.Collection<E>`
- metodi

`static <E> Set<E> of(E... elements)` → *factory method*
(simile al metodo *List.of(elements)* di *List*)

```
import java.util.HashSet<E>;
```

- metodi ereditati da `java.util.Set<E>`
- metodi

`HashSet(Collection<? extends E> parent)` → crea un insieme di tipo *HashSet* e lo riempie con gli elementi di *parent*

Insieme ordinato → implementa *Comparable<T>* (*CompareTo(T other)* e *equals(Object other)*) o prende in input un *Comparator*

Implementa anche un suo *hashCode()* o utilizza quello di *obj*

```
import java.util.SortedSet<E>;
```
interfaccia

- metodi ereditati da `java.util.Set<E>`
- metodi

`E first() throws java.util.NoSuchElementException` → ritorna il primo elemento (il più piccolo) dell'insieme

`E last() throws java.util.NoSuchElementException` → ritorna l'ultimo elemento (il più grande) dell'insieme

```
import java.util.TreeSet<E>;
```

- metodi ereditati da `java.util.SortedSet<E>`
- metodi

`TreeSet(Collection<? extends E> parent)` → crea un insieme di tipo *TreeSet* e lo riempie con gli elementi di *parent*

`import java.util.Map<K,V>;` interfaccia

- metodi ereditati da `java.util.Collection<E>`
- metodi

`boolean containsKey(Object key)` → controlla se la mappa contiene la chiave passatagli
`boolean containsValue(Object value)` → controlla se la mappa contiene il valore passatogli
`V get(Object key)` → data la chiave, restituisce il valore (se non c'è la chiave ritorna null)
`V getOrDefault(Object key, V defaultValue)` → prende in input chiave e valore di *default*, come *get(key)*, ma se non trova la chiave, invece di ritornare *null*, ritorna il valore di *default*
`boolean isEmpty()` → controlla se la mappa è vuota
`Set<K> keySet()` → ritorna l'insieme delle chiavi (utilizzato nei *for each*)
`V put(K key, V value)` → mette nella mappa un nuovo *chiave : valore* se assente, (ritorna *null*), sennò sostituisce il valore corrisponde alla chiave, se quest'ultima già esiste, (ritorna il vecchio valore)
`V putIfAbsent(K key, V value)` → mette nella mappa un nuovo *chiave : valore* se assente, (ritorna *null*), se già presente ritorna il vecchio valore
`V remove(Object key)` → rimuove il valore corrispondente a *key*, se presente, e lo ritorna, se non c'è ritorna *null*
controlla che elemento rimuovere per il suo *equals*
(riscrivere *equals* → *equals* di *obj* (==))
`Collection<V> values()` → ritorna una *Collection* del tipo delle chiavi
(dopo bisogna passarla a una struttura dati concreta)

`import java.util.HashMap<K,V>;`

- metodi ereditati da `java.util.Map<K,V>`
- metodi

`HashMap(Map<? extends K,? extends V> parent)` → crea una mappa di tipo *HashMap* e la riempie con le coppie *chiave : valore* contenute in *parent*

Mappa ordinato → implementa *Comparable<T>* (*CompareTo(T other)* e *equals(Object other)*)
o prende in input un *Comparator*

Implementa anche un suo *hashCode()* o utilizza quello di *obj*

`import java.util.SortedMap<K,V>;` interfaccia

- metodi ereditati da `java.util.Map<K,V>`
- metodi

`K firstKey()` → ritorna la prima (la più piccola) chiave della mappa
`K lastKey()` → ritorna l'ultima (la più grande) chiave della mappa

`import java.util.TreeMap<K,V>;`

- metodi ereditati da `java.util.SortedMap<K,V>`
- metodi

`TreeMap(Map<? extends K,? extends V> parent)` → crea una mappa di tipo *TreeMap* e lo riempie con le coppie *chiave : valore* contenute in *parent*

`import java.util.Queue<E>;` interfaccia

- metodi ereditati da `java.util.Collection<E>`
- metodi

`E poll()` → rimuove la testa della coda e la ritorna (se la coda è vuota ritorna *null*)

`E remove() throws java.util.NoSuchElementException` → rimuove la testa della coda e la ritorna, se è vuota lancia un'eccezione

`E peek()` → ritorna la testa della coda (se la coda è vuota ritorna *null*)

`E element() throws java.util.NoSuchElementException` → ritorna la testa della coda, se è vuota lancia un'eccezione

`boolean offer(E element)` → aggiunge *element* in fondo alla coda, se c'è spazio. ritorna *true* sse *element* viene aggiunto

`boolean add(E element) throws java.lang.IllegalStateException` → aggiunge *element* in fondo alla coda, se c'è spazio, se non c'è spazio lancia un'eccezione. ritorna sempre *true*

coda unbounded → non ha un limite massimo di elementi (si espande quando necessario)

`import java.util.PriorityQueue<E>;`

- metodi ereditati da `java.util.Queue<E>`
- metodi

`TreeSet(Collection<? extends E> parent)` → crea una coda di tipo *PriorityQueue* e la riempie con gli elementi di *parent*

`import java.lang.Iterable<T>;` interfaccia
`Iterator<T> iterator()` → ritorna un iteratore, cioè un oggetto capace di restituire gli elementi dell'iterabile, uno alla volta (*if(hasNext) {return next}*)

`import java.util.Iterator<T>;` interfaccia
`boolean hasNext()` → controlla se ha un *next* `T next()` → ritorna l'elemento *next* (di solito di una *Collection*)

interfaccia java.lang.Iterable<E>

Mettere esempi costruzione

Da zero

Chiamando l'iterator della superclasse

Passando per altre strutture dati e prendendo il loro iteratore

Utilizzo

Passare in input l'implementazione di un o più metodi (con un nome) derivanti da un'interfaccia, senza però specificare il nome della classe.
Utilizzate in più punti della stessa classe.

```
Nome_interfaccia<T> nome_segnaposto = new Nome_interfaccia<T>() {
      @Override
      … nome_metodo_da_implementare(input) {      // firma metodo da implementare
            codice
      }
};
```

```
Predicate<Studente> p = {
    @Override
    public boolean test(Studente studente) {
       return studente.fuoriCorso(informatica);
    }
};
```

```
Consumer<Studente> c = {
    @Override
    public void accept(Studente studente) {
       System.out.println(studente.getMatricola());
    }
};
```

Usate per passare come parametro in input
l'implementazione di un metodo di un'interfaccia,
senza dover dichiarare nuove classi o metodi.
Di solito utilizzate una volta.

```
input -> (return implicito) codice_nella_riga;
```

```
input -> {
    codice
}
```

```
(input1, input2) -> (return implicito) codice_nella_riga;
```

```
(input1, input2) -> {
    codice
}
```

`import java.util.function.Consumer<T>`	interfaccia
`void accept(T t)` → esegue del codice che usa *t*	

`import java.util.function.Predicate<T>`	interfaccia
`boolean test(T t)` → determina se *t* soddisfa il predicato (per verificare se una certa condizione di una proprietà è soddisfatta)	

`import java.util.function.Supplier<T>`	interfaccia
`T get()` → ritorna un oggetto di tipo *T*	

`import java.util.function.Function<T>`	interfaccia
`boolean U apply(T t)` → ritorna il valore della funzione applicata a *t*	

```
Predicate<Studente> p = {
    @Override
    public boolean test(Studente studente) {
       return studente.fuoriCorso(informatica);
    }
};

Consumer<Studente> c = {
    @Override
    public void accept(Studente studente) {
       System.out.println(studente.getMatricola());
    }
};

esame.perOgniIscritto(p,c);
```

Utilizzo

```
public void perOgniIscritto(Predicate<Studente> condizione,
                            Consumer<Studente> azione) {
    for (Studente s : iscritti) {
       if (condizione.test(s))
          azione.accept(s);
    }
}
```