**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)

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)); |

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); Scanner(source)  int numero = scanner.nextInt(); |

Stringhe

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()* (?) |

Mettere esempi

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() |

Mettere esempi

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

Array

Mettere esempi

Dichiarazione array

Inizializzato

Dimensione inizializzata

Fuori dal metodo

Static

new

Utilizzo

Length

For each

Inizializzazione for normale

Altro

L’esempio della matrice, array di array (SoccerPlayer)

Ritorna puntatore

|  |
| --- |
| import java.util.Arrays; |
| static int binarySearch(int[] arr, int key) → ritorna la posizione di key dentro arr,  oppure 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 per i tipi riferimento, nel qual caso 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 nonché per array di tipi riferimento, nel qual caso 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 per array di tipi riferimento.  static void sort(int[] arr) → ordina arr in senso crescente, in tempo O(n log n).  Questo metodo esiste anche per tutti gli altri tipi primitivi numerici e per i tipi riferimento, nel qual caso chiama compareTo() per decidere l'ordine.  static String toString(int[] arr) → ritorna una stringa che riporta gli elementi di arr, nel loro ordine. Questo metodo esiste anche per gli altri tipi primitivi e per array di tipi riferimento, nel qual caso chiama *toString()* sugli elementi dell'array e concatena il risultato. |

Mettere esempi classe Arrays

Costruttore

Mettere esempi

Costruttore

Costruttore vuoto

Più costruttori con stesso nome

Utilizzo

Enumerazione

Classe di costanti

Mettere esempi

Classe enum

Utilizzo

Come chiamare classe

Alti esempi, tipo ordinal()

**Metodi di uso frequente delle classi E definite tramite enum**

*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) → determina chi viene prima nell'enumerazione

int ordinal() → ritorna il numero d'ordine di un elemento dell'enumerazione

Ereditarietà

Extends

Superclasse - sottoclasse

Super

Costruttori non si ereditano

super()

super.

@Override

Sovrascrive il metodo della superclasse con la stessa firma

(tipo di ritorno – nome metodo – lista di input (quantità, tipo e ordine)),

quando il metodo viene chiamato, viene chiamato il metodo della classe più vicina

sottoclasse può implementare suoi metodi

se non si fa l’@Override, la classe eredita i metodi public (o protected se nella stessa cartella),

i metodi private ci sono ma non si vedono

java può estendere una sola classe ma implementare più interfacce

riceve in input l’interfaccia o la superclasse, gli viene passata la sottoclasse che può usare solo metodi definiti della superclasse, perché di tipo della superclasse, ma usa l’implementazione della sottoclasse (classe più vicina). Per ovviare si usa il casting dopo aver controllato (instanceof)

private non può essere usato al di fuori del file in cui si trova

interfaccia → implements

classi → extends

toString()

equals() e @Override equals()

java.lang.Object

equals() e compareTo() implements Comparable<T>

Interfacce

Vengono specificati i metodi da implementare nella sua sottoclasse

La visibilità delle interfacce e dei loro metodi è implicitamente public.

Classi abstract

Ibrido tra classe concreta e interfaccia, contiene sia metodi già implementati,

sia metodi da implementare, contrassegnati con abstract

Parametri varargs

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

Vincoli sulle variabili di tipo ???

|  |
| --- |
| import java.lang.Comparable<T>; |

int compareTo(T other) → ritorna un numero negativo se viene prima this, un numero positivo se viene prima other e 0 se this e other sono equivalenti.

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)* |

Eccezioni

schema eccezioni (tablet → gerarchia delle eccezioni)

throw new eccezione()

throws per costruttori e metodi

try/catch/finally

definizione di nostre classi di eccezione

costruttore senza input

costruttore con input

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamenteI/O

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"); } |

Immagine che contiene testo, schermata, cerchio, Carattere

Descrizione generata automaticamente

deprecata!

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)); } |

Liste

|  |
| --- |
| 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) |

Insiemi

|  |
| --- |
| 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* |

Mappe

|  |
| --- |
| 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* |

Code

|  |
| --- |
| 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* |

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

Le interfacce di libreria java.lang.Iterable<T> e java.util.Iterator<T>. Loro implementazione per rendere iterabili le nostre classi. Esempio di implementazione di iterator() per delega, per il caso del Desk del laboratorio del 6 dicembre. Classi interne e classi interne statiche. Esempio di implementazione con una classe interna.

Metodi di uso frequente dell'interfaccia java.lang.Iterable<T>

Iterator<T> iterator() (ritorna un iteratore, cioè un oggetto capace di restituire gli elementi dell'iterabile, uno alla volta)

Metodi di uso frequente dell'interfaccia java.util.Iterator<T>

boolean hasNext()

T next()

Classi anonime. Lambda espressioni. Interfacce funzionali. Riferimenti a metodi e costruttori. Uso delle lambda espressioni per iterazione interna su collezioni.

Metodi di uso frequente dell'interfaccia java.util.function.Consumer<T>

void accept(T t) (esegue del codice che usa t)

Metodi di uso frequente dell'interfaccia java.util.function.Predicate<T>

boolean test(T t) (determina se t soddisfa il predicato)

Metodi di uso frequente dell'interfaccia java.util.function.Supplier<T>

T get() (fornisce un oggetto di tipo T)

Metodi di uso frequente dell'interfaccia java.util.function.Function<T,U>

boolean U apply(T t) (ritorna il valore della funzione applicata a t)