Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Strutture dati: il Collection Framework

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

COLLECTION FRAMEWORK

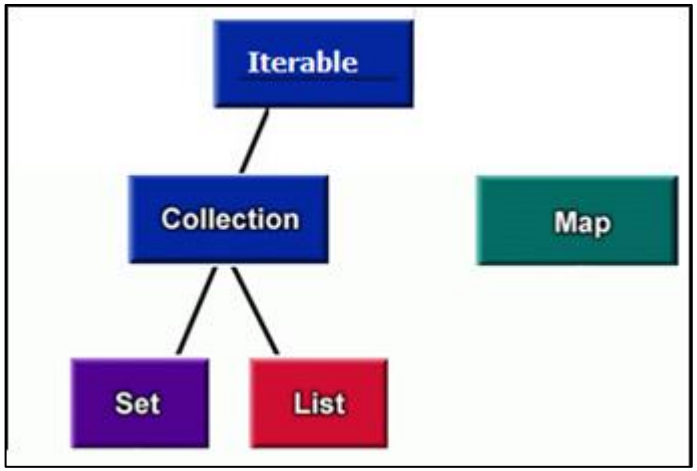
- Poiché le strutture dati giocano un ruolo essenziale in qualunque applicazione non banale, tutti i linguaggi a oggetti più diffusi definiscono un proprio *Collection Framework*
  - una *architettura logica globale e uniforme* per le strutture dati
  - un insieme di interfacce & classi che definiscono e implementano le strutture dati più frequentemente utilizzate, coi relativi algoritmi
- Obiettivo: strutture dati *generiche,* parametriche in tipo
  - interfacce che introducono i *tipi di strutture dati* (liste, set, mappe) con i necessari *concetti di supporto* (entità iterabili & iteratori)
  - classi che forniscono *una o più implementazioni* di tali concetti
  - eventuali *librerie accessorie* (se necessario) con funzioni statiche che incapsulano *algoritmi polimorfi*, costanti di uso generale, etc.

COLLECTION FRAMEWORK

- Elementi comuni nei diversi linguaggi
  - concetti come liste, set, mappe, iteratori
  - array come *tipi a sé stanti,* non come particolari collection ☹
  - strutture dati *parametriche rispetto al tipo*
- Elementi di differenza fra i diversi linguaggi
  - collezioni sempre modificabili o, al contrario, *distinzione* fra collezioni immutabili vs. modificabili
  - tipo parametrico ristretto a oggetti (quindi *non tipi primitivi Java*) o in grado di coprire qualunque tipo (C#, Scala, Kotlin)
  - tipo parametrico esistente *solo a compile-time* (*type erasure*: Java, Scala, Kotlin) o *reificato anche a run time* (C#, .NET in generale)

OVERVIEW GENERALE

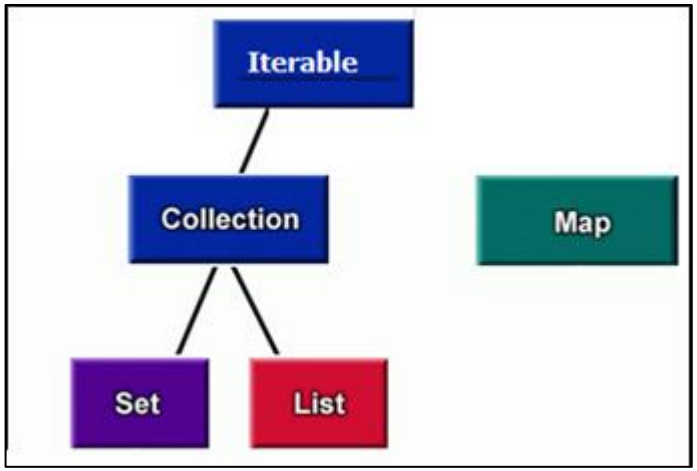
- Praticamente tutti i linguaggi offrono:
  - «collezione»: insieme di elementi senza nessuna ipotesi particolare
  - «set»: introduce l'idea di *insieme* di elementi, *senza duplicati*
  - «lista» o «sequenza»: introduce l'idea di *sequenza* di elementi
  - «mappa»: introduce l'idea di *tabella* che associa chiavi a valori
  - …e varianti



- Gli array non fanno parte delle collection ☹
  - sono forniti metodi di conversione array ↔ collections
  - ma l'ideale è smettere proprio di usare gli array…!

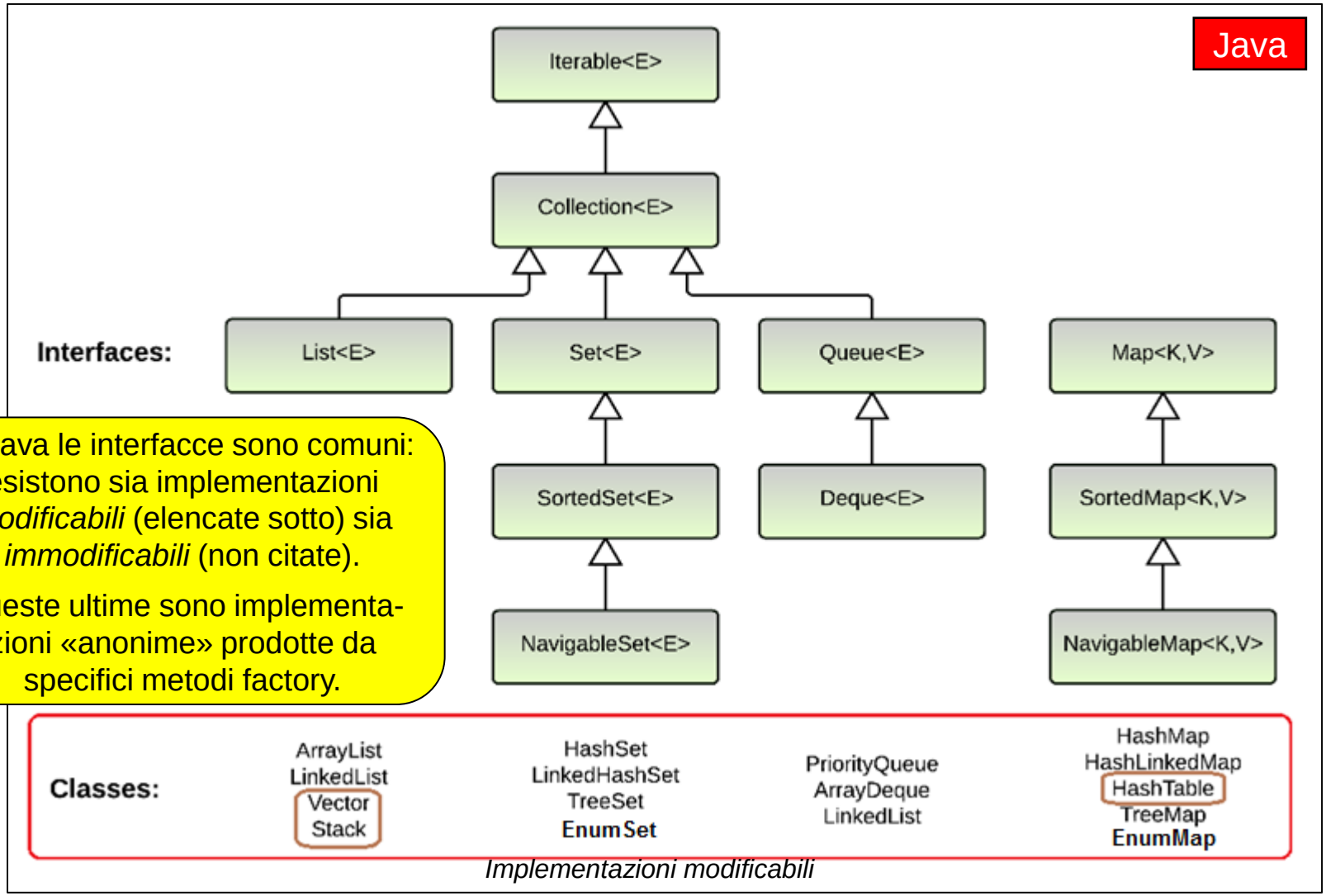
CONVENZIONI DI NAMING

- Nei linguaggi di derivazione Java (Scala, Kotlin)
  - i nomi «generali» dei concetti a lato denotano interfacce → *il front-end* (spesso, nella variante immodificabile)
  - le implementazioni del back-end hanno invece nomi più specifici (es. `HashSet`, `ArrayList`)



- In C# e nei linguaggi .NET
  - i nomi «più ovvi» dei concetti a lato denotano implementazioni (classi)
  - le interfacce (front-end) hanno invece nomi della forma `I`*`Qualcosa`*, come `ICollection`, `IList`, etc.

COLLECTION FRAMEWORK: JAVA



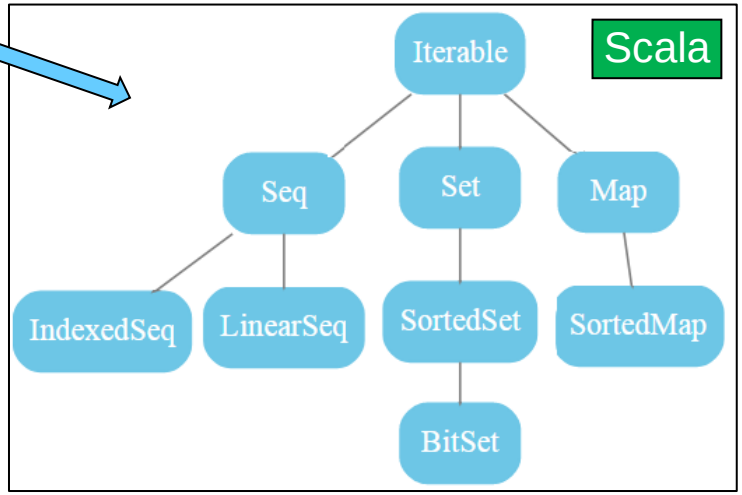
In Java le interfacce sono comuni: esistono sia implementazioni *modificabili* (elencate sotto) sia *immodificabili* (non citate).

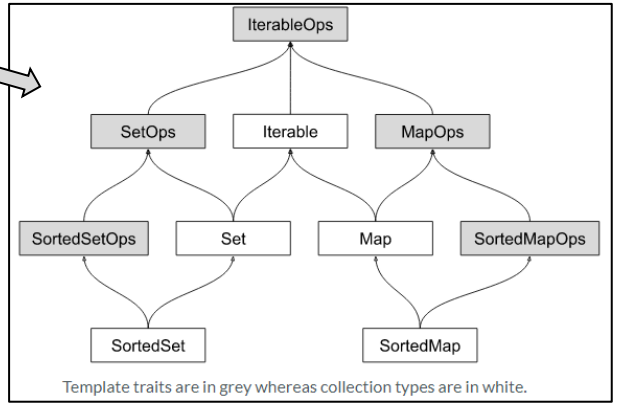
Queste ultime sono implementazioni «anonime» prodotte da specifici metodi factory.

Implementazioni modificabili

COLLECTION FRAMEWORK: SCALA

- Scala separa collezioni immutabili vs. modificabili
  - tratti top-level, validi sia per collezioni immutabili che modificabili, catturano le *tipologie generali di collezioni* (set, sequenze, mappe)
  - tratti più specifici, *distinti per collezioni immutabili vs. modificabili*, specializzano i due casi
  - ulteriori tratti (contenenti codice) fattorizzano operazioni comuni
  - infine, un'*ampia serie di classi* fornisce le implementazioni del back-end, distinte per collezioni immutabili vs. modificabili

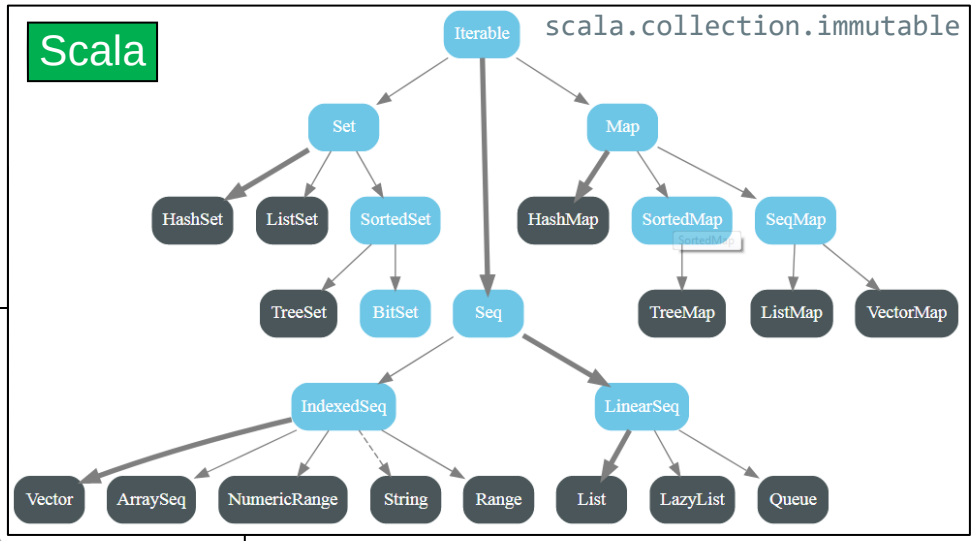


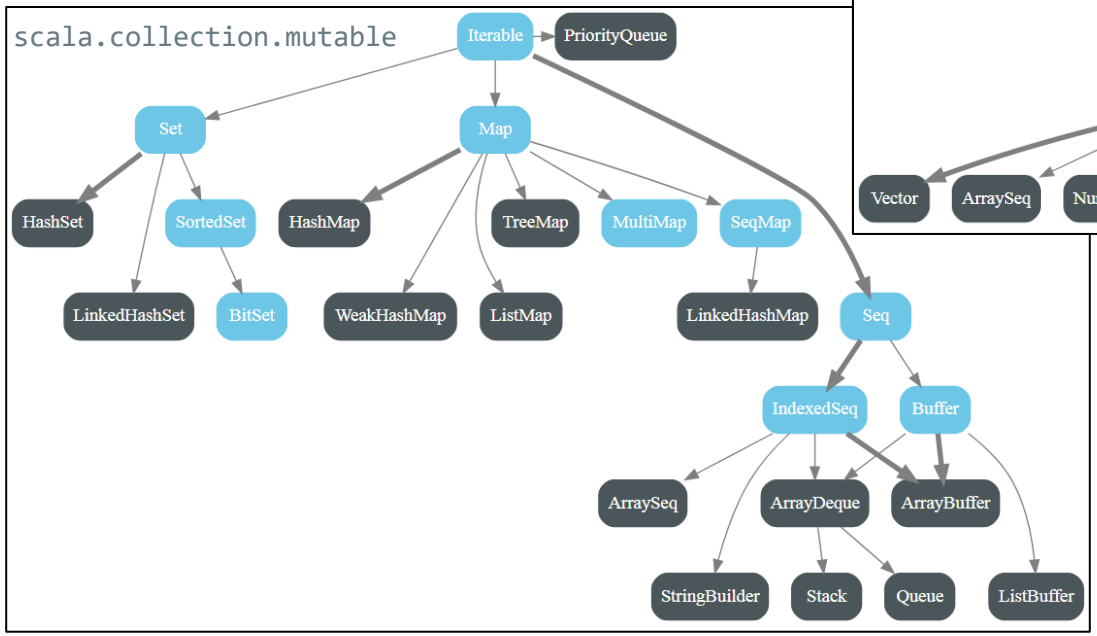


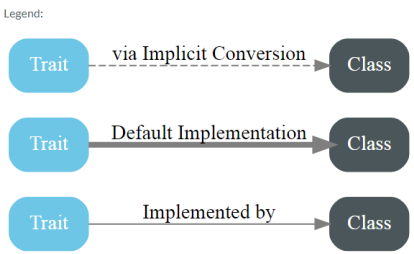
COLLECTION FRAMEWORK: SCALA

- Scala separa collezioni immutabili vs. modificabili in *due package distinti*

NB: questi diagrammi non seguono lo standard UML

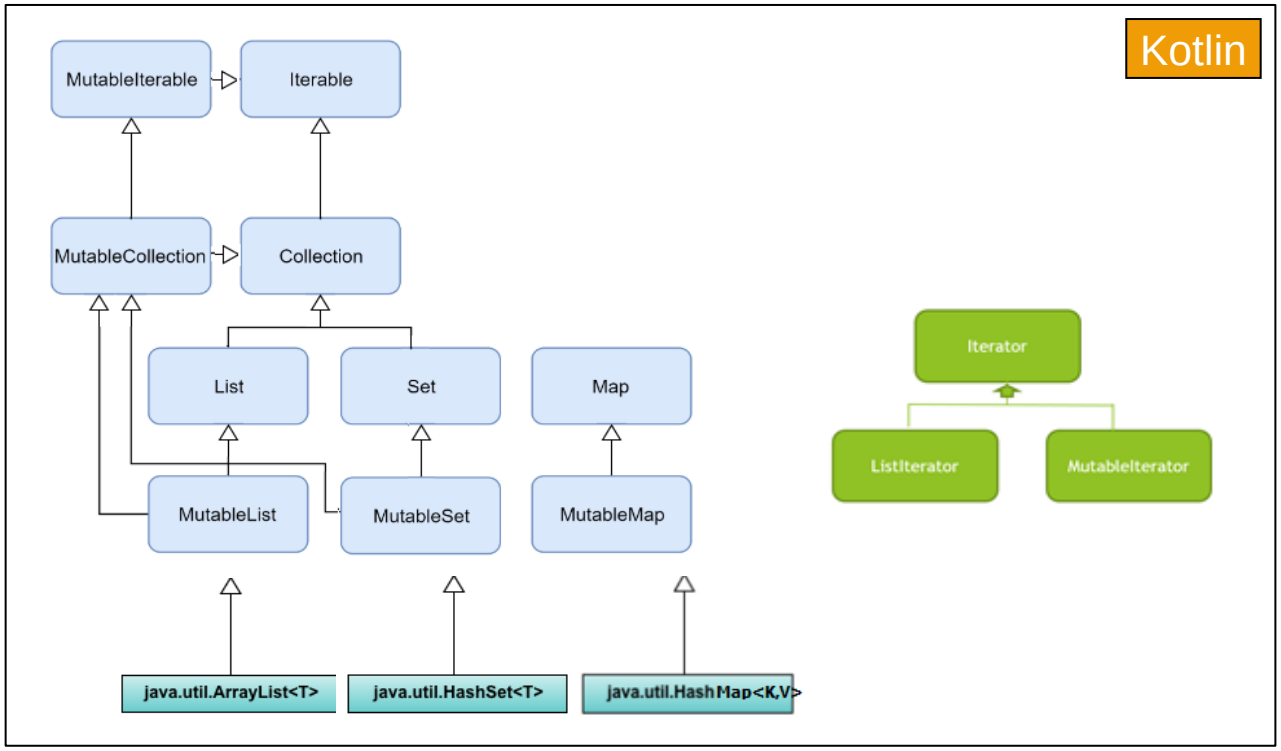






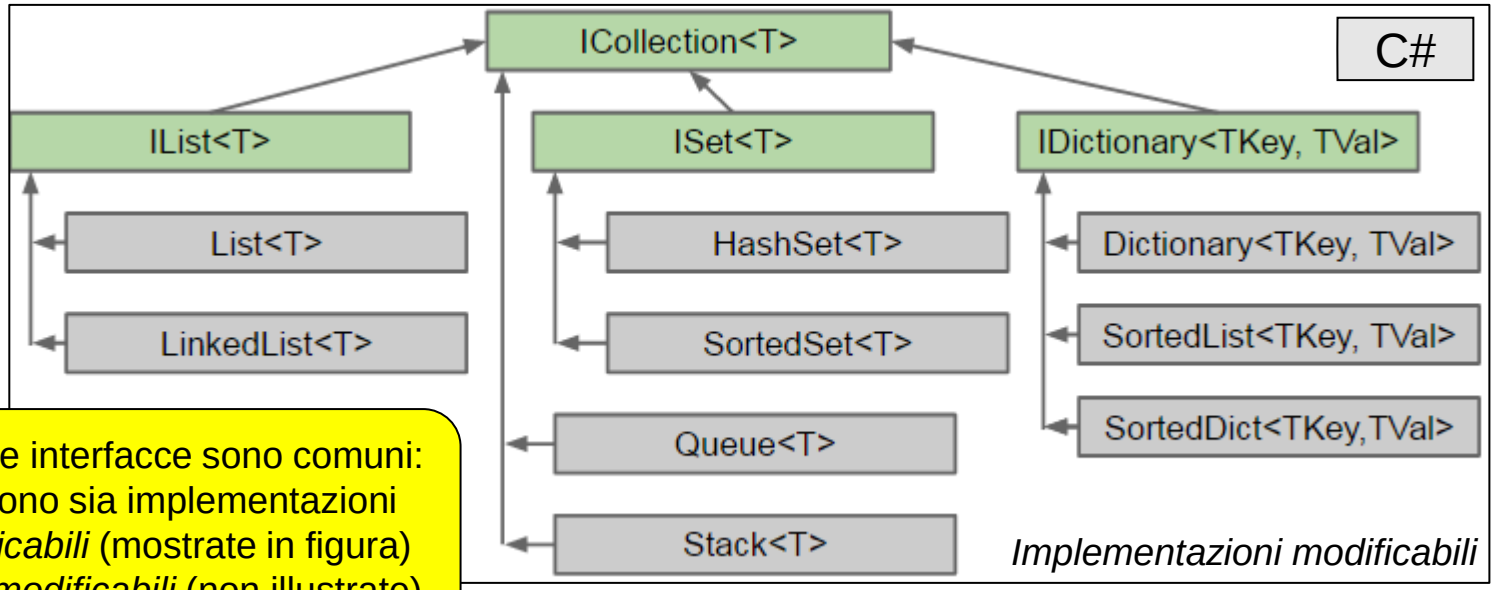
COLLECTION FRAMEWORK: KOTLIN

- Anche Kotlin separa collezioni immutabili vs. modificabili
  - serie di interfacce `Mutable*` che derivano da quelle «non-mutable»
  - classi del back-end (Java-like/Java) che le implementano



COLLECTION FRAMEWORK: C#

- C# definisce interfacce che originano da `IEnumerable<T>`
  - analogo a `Iterable` di Java
- Struttura generale simile a quella di Java
  - da `IEnumerable<T>` discende `ICollection<T>`, etc.



In C# le interfacce sono comuni: esistono sia implementazioni *modificabili* (mostrate in figura) sia *immodificabili* (non illustrate)

COLLECTION FRAMEWORK: CROSS-LANGUAGE OVERVIEW

- Principali interfacce e classi nei diversi linguaggi
  - in Scala, di default, è importata la versione *immutable*

		Set		List		Map	
		Mutable	*Immutable*	*Mutable*	*Immutable*	*Mutable*	*Immutable*
Java	*interfaces*	`Set`	`Set`	`List`	`List`	`Map`	`Map`
	classes	`HashSet` `TreeSet`	(Set.of)	`ArrayList` `LinkedList`	(List.of)	`HashMap` `TreeMap`	(Map.of)
C#	*interfaces*	`ISet`	`IImmutableSet`	`IList`	`IImmutableList`	`IDictionary`	`IImmutableDictionary`

	classes	`Set`
		`HashSet`
		`SortedSet`
			`ImmutableSet`
			`ImmutableHashSet`
				`List`
				`SortedList`
					`ImmutableList`	`Dictionary`
						`SortedDictionary`
						`SortedList`
							`ImmutableDictionary`
							`ImmutableSortedDictionary`

Scala	*traits*	`Set`	`Set`	`Seq, Buffer`	`Seq, LinearSeq`	`Map`	`Map`

	classes	`HashSet`
		`LinkedHashSet`
			`HashSet`
			`ListSet`
			`TreeSet`
				`ArrayBuffer`
				`ListBuffer`
					`List`
					`LazyList`
					`Queue`
						`HashMap`
						`TreeMap`,
						`ListMap`, …
							`HashMap`
							`TreeMap`,
							`ListMap`, …

Kotlin	*interfaces*	`MutableSet`	`Set`	`MutableList`	`List`	`MutableMap`	`Map`

	classes	`HashSet`	`HashSet`	`ArrayList`	`ArrayList`	`HashMap`
						`TreeMap`
						`ListMap`, …
							`HashMap`
							`TreeMap`
							`ListMap`, …

OBIETTIVO: GENERICITÀ

- Le strutture dati sono *contenitori* per elementi *di diversi tipi*
  - liste di interi, code di Persone, liste di Frazioni…
- ..MA naturalmente si vuole definirle *una volta sola*
  - indipendentemente dal tipo degli oggetti che conterranno
  - liste di "quello che voglio", code di "quel che mi serve", etc.
- La *genericità* è il mezzo per ottenere ciò
  - si vogliono *liste, code, alberi.. generici rispetto al tipo*
- Si può ottenere in modi diversi
  - approccio antico: polimorfismo verticale → sfruttare `Object`
  - approccio moderno: polimorfismo orizzontale → usare *tipi generici*

L'APPROCCIO ANTICO

- In Java ≤ 1.4 e C# originale, le collection adottavano **`Object`** (C#: **`object`**) come *tipo dell'elemento*
  - pro: poiché tutto (in Java: tranne i tipi primitivi) deriva da `Object`, ciò consentiva effettivamente di *fare strutture di «quel che si voleva»*
  - contro: proprio poiché (quasi) tutto deriva da `Object`, tutto era (quasi) compatibile con tutto → *le strutture non potevano garantire omogeneità di tipo degli elementi: ci finiva letteralmente dentro «di tutto»*
- Tale approccio non era «type safe»
  - errori logici non potevano essere svelati dal compilatore
  - necessità di *continui cast* per inserire/estrarre oggetti dalle collezioni

Un approccio obsoleto, ormai non più usato da anni

L'APPROCCIO ANTICO: PERCHÉ NO

- Nel primo esempio, `list` è una lista di `Object` → ci si può mettere dentro di tutto.. ma bisogna trattarli «da Object», senza fare altre assunzioni
- Nel secondo esempio, l'intenzione era quella di avere una lista «di `Counter`», ma nei fatti è comunque di `Object` → ci si può *ancora* mettere dentro di tutto, ma si viola un vincolo semantico

C#

```
public static void Main()
{
    System.Collections.IList list = new System.Collections.ArrayList();
    list.Add(new Counter(10));
    list.Add("ciao");
    foreach(object obj in list) Console.WriteLine(obj);

    System.Collections.IList listOfCounters = new System.Collections.ArrayList();
    listOfCounters.Add(new Counter(11));
    listOfCounters.Add("ciao");
    listOfCounters.Add(new Counter(7));
    // foreach(Counter u in listOfCounters) Console.WriteLine(u); // NO: incompatible types
    Counter c0 = (Counter) listOfCounters[0]; Console.WriteLine(c0);
    Counter c2 = (Counter) listOfCounters[2]; Console.WriteLine(c2);
    Counter cx = (Counter) listOfCounters[1]; Console.WriteLine(cx); // boom
```

```
10
ciao
11
7
Run-time exception (line 19): Unable to cast object of type 'System.String' to type 'Counter'.
```

```
public static void main(String[] args){
   java.util.List list = new java.util.ArrayList();
   list.add(new Counter(10));
   list.add("ciao");
   for(Object obj : list) System.out.println(obj);

   java.util.List listOfCounters = new java.util.ArrayList();
   listOfCounters.add(new Counter(11));
   listOfCounters.add("ciao");
   listOfCounters.add(new Counter(7));
   // for(Counter u : listOfCounters) System.out.println(u); // NO: incompatible types
   Counter c0 = (Counter) listOfCounters.get(0); System.out.println(c0);
   Counter c2 = (Counter) listOfCounters.get(2); System.out.println(c2);
   Counter cx = (Counter) listOfCounters.get(1); System.out.println(cx); // boom
```

Java

```
10
ciao
11
7
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to Counter
        at HelloWorld.main(HelloWorld.java:22)
```

Conseguenza: quando si va a estrarre un elemento ritenendo (erroneamente) che sia un `Counter`, il cast fallisce e…

L'APPROCCIO ANTICO: PERCHÉ NO

- Il problema di questo approccio è che il design intent del progettista rimane *formalmente inespresso*
  - al più è affidato a qualche «bel nome» (**`listOfCounters`**) o a qualche commento → l'anticamera del disastro
- Il punto chiave è che il vincolo sul tipo degli oggetti contenuti, che il progettista aveva in mente, non si è trasferito nel codice
  - il fatto che **`listOfCounters`** volesse essere una lista di **`Counter`** è affidato *solo al nome scelto*, che ovviamente non ha significato per il compilatore: non a qualcosa di *linguistico* percepibile dal compilatore
  - per il type system **`listOfCounters`** è solo una normalissima lista di **`Object`**.. ma così, l'errore logico non viene intercettato, e al momento del cast… esplode tutto a run time ☹

L'APPROCCIO ANTICO: PERCHÉ NO

- Usare il tipo `Object` per fare strutture dati generiche equivale nei fatti ad *abolire il controllo di tipo*
  - la correttezza è affidata solo a "commenti sul corretto uso", anziché ai controlli del compilatore → *l'anticamera del disastro*
- Ergo, operazioni *sintatticamente corrette* possono risultare in realtà *semanticamente errate* → errori inattesi a runtime
- MORALE:
  un approccio basato su **`Object`** non è «type safe»
- Per avere codice robusto il controllo di tipo dev'essere *rafforzato,* non eluso!

L'APPROCCIO ANTICO: TIPI «RAW»

- L'approccio antico rimane in Java e C# per retrocompatibilità, ma è *deprecato* da anni
  - si usa dire che la vecchia versione utilizza *raw types* (tipi grezzi) e *unchecked operations* (operazioni non controllate)
  - compilandola si ottengono warning con invito e sostituirla

```
Note: TestBoxing.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

- Nel seguito useremo solo la versione moderna
  - però, in Java, essa è solo *un'illusione ottica del compilatore*: sotto sotto, nella JVM si usa ancora `Object` (tecnica di *type erasure*) … anche se non ve ne accorgerete (quasi) mai ☺
  - in C#, viceversa, i tipi generici sono effettivamente *reificati a run-time*

L'APPROCCIO MODERNO

- Occorre rovesciare il punto di partenza:
  è intrinsecamente sbagliato abolire il controllo di tipo
  - si deve rafforzarlo, non indebolirlo, né tantomeno eluderlo!
  - Obiettivo TYPE SAFETY:
    "se si compila, l'uso dei tipi è certamente corretto"
- La genericità è necessaria, *ma nel modo giusto,*
  che non può essere «usare **`Object`** ovunque»
- Per questo, l'approccio moderno adotta a tappeto il concetto di *tipo parametrico (generico)*
  - un modo pulito per esprimere genericità in tipo
  - senza abusare del povero `Object`

L'APPROCCIO MODERNO

- Con il *tipo generico* **<T>** la strutture sono *tipizzate*
  - non più strutture «di qualunque cosa», ma *di quel certo tipo* **<T>**
  - si garantisce omogeneità di tipo degli elementi contenuti
    → *non ci può più finire dentro «di tutto»*
- Risultato: «type safety»
  - errori logici vengono svelati dal compilatore
  - si evitano i cast prima per inserire/estrarre oggetti dalle collezioni
- È ormai da anni l'approccio standard, da usare sempre
  - in Scala e Kotlin è anche *l'unico disponibile*

TIPI GENERICI

- Per esprimere l'idea che una certa entità è *generica rispetto ai tipi che manipola* si adotta la notazione

  `<TIPO>` (in Scala: **`[Tipo]`**)

- Le strutture dati (o le funzioni) che manipolano quei tipi vengono *esplicitamente taggate* con quell'etichetta
  - il compilatore può così effettuare *controlli stringenti* di tipo per garantire *TYPE SAFETY:* è un *potente alleato*, non un nemico!

- Non più:

```
List listOfCounters = … // lista di counter
List listOfPersons = …  // lista di persone
```

- bensì:

```
List<Counter> listOfCounters = …
List<Persona> ListOfPersons = …
```

Java C# ~Scala ~Kotlin

L'APPROCCIO MODERNO: PERCHÉ SÌ

- A differenza di prima, ora l'intenzione di avere una lista «di **`Counter`**» è *reificata nel codice* → il vincolo semantico è *esplicitamente espresso*
- Non è più possibile cercare di inserire qualcosa di diverso, perché l'errore viene subito intercettato dal compilatore
- Ulteriore vantaggio: essendo noto il tipo specifico, *non serve più il cast* ☺

C#

```
public static void Main()
{

    System.Collections.Generic.IList<Counter> listOfCounters = new System.Collections.Generic.List<Counter>();
    listOfCounters.Add(new Counter(11));
    //listOfCounters.Add("ciao"); // NO: cannot convert from string to Counter
    listOfCounters.Add(new Counter(7));
    Counter c0 = listOfCounters[0]; Console.WriteLine(c0);
    Counter c1 = listOfCounters[1]; Console.WriteLine(c1);


11
7
```

Notazione shortcut (Diamond operator)

Java

```
public static void main(String[] args){

   java.util.List<Counter> listOfCounters = new java.util.ArrayList<>();
   listOfCounters.add(new Counter(11));
   //listOfCounters.add("ciao"); // NO: no suitable method for add(String)
   listOfCounters.add(new Counter(7));
   Counter c0 = listOfCounters.get(0); System.out.println(c0);
   Counter c1 = listOfCounters.get(1); System.out.println(c1);
```

```
11
7
```

L'APPROCCIO MODERNO: PERCHÉ SÌ

- Come si diceva, in Scala e Kotlin l'approccio moderno è l'unico disponibile
- Di più: *distinguendo fra collezioni modificabili vs. immodificabili*, ogni tentativo di aggiungere/togliere elementi a una struttura *immodificabile* viene stroncato

Scala

```scala
import scala.collection.mutable._

object GenericCollectionsTest{

  def main(args: Array[String]) : Unit = {
    val listOfCounters : Buffer[Counter] = new ListBuffer[Counter]();
    listOfCounters += new Counter(11);
    // listOfCounters += "ciao"                    // type mismatch: found String("ciao"), required Counter
    listOfCounters += new Counter(7);
    val c0 = listOfCounters(0); println(c0)        // Counter di valore 11
    val c1 = listOfCounters(1); println(c1)        // Counter di valore 7

    val immutableListOfCounters = List(c1,c0);
    println(immutableListOfCounters)               // List(Counter di valore 7, Counter di valore 11)
    // immutableListOfCounters += new Counter(3)   // value += is not a member of List[Counter]
  }

}
```

```
Counter di valore 11
Counter di valore 7
List(Counter di valore 7, Counte
```

Kotlin

```kotlin
import kotlin.collections.*

fun main() {
    val listOfCounters : MutableList<Counter> = ArrayList<Counter>();
    listOfCounters.add(Counter(11));
    //listOfCounters.add("ciao")                // type mismatch: inferred type is String but Counter was expected
    listOfCounters.add(Counter(7));
    val c0 = listOfCounters.get(0); println(c0) // Counter di valore 11
    val c1 = listOfCounters.get(1); println(c1) // Counter di valore 7

    val immutableListOfCounters = listOf(c1,c0);
    println(immutableListOfCounters)           // [Counter di valore 7, Counter di valore 11]
    // immutableListOfCounters.add(Counter(3)) // Unresolved reference: add
}
```

```
Counter di valore 11
Counter di valore 7
[Counter di valore 7, Counter di valore 11]
```

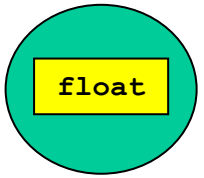
Java: wrapper per tipi primitivi

UN PROBLEMA: I TIPI PRIMITIVI

- A differenza di C#, Scala e Kotlin, in Java i tipi primitivi non sono veri oggetti
  - di conseguenza, richiedono sempre un trattamento ad hoc
  - non hanno metodi: le operazioni sono funzioni statiche di libreria
  - non sono uniformi agli altri oggetti: usano == invece di equals, adottano il passaggio per valore anziché per riferimento, hanno tipi-array separati, etc.
- Soprattutto, non derivando da **`Object`**, non si possono usare direttamente nelle collection
  - non si può scrivere **`List<int>`**, **`Set<long>`**, etc.
  - non a caso, tutti i linguaggi più recenti eliminano i tipi primitivi sostituendoli con vere classi *(«value classes»)*

UN PROBLEMA: I TIPI PRIMITIVI

- La soluzione Java per garantire interoperabilità si basa sulla definizione di opportune *classi wrapper*
- Ogni *wrapper class* incapsula *un valore di tipo primitivo*
- Di base, la *wrapper class* è «quasi omonima» al corrispondente tipo primitivo…
  - classe **`Long`** per incapsulare il tipo primitivo **`long`**
  - classe **`Float`** per incapsulare il tipo primitivo **`float`**
  - classe **`Double`** per incapsulare il tipo primitivo **`double`**
  - …etc.
- .. ma ci sono alcune eccezioni

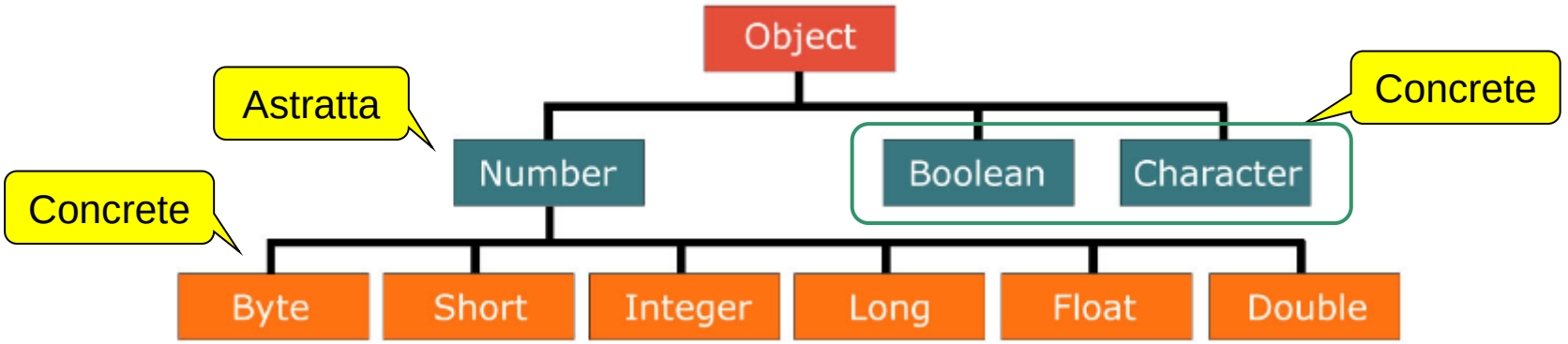


`Float`

CLASSI WRAPPER IN JAVA

- Ogni tipo primitivo ha una classe wrapper *«quasi omonima»*
  - di norma, stesso nome con iniziale maiuscola
  - tranne i due casi `int` → `Integer`, `char` → `Character`

Java



- Le classi wrapper numeriche derivano dalla classe astratta **`Number`**, che definisce svariati metodi di conversione
  - `intValue`, `longValue`, `floatValue`, `doubleValue`, etc.

CLASSI WRAPPER IN JAVA: BOXING & UNBOXING

- L'operazione di incapsulamento di un valore primitivo nel corrispondente oggetto wrapper si chiama BOXING
  - fino a Java 8, si faceva tramite il costruttore del wrapper

```
Integer i = new Integer(22);
Double  d = new Double(3.14);
```

Java

  - da Java 9 tale approccio è deprecato: il costruttore pubblico sarà presto *rimosso,* nella direzione di avere vere *value-based classes*
  - nuovo approccio: costruzione tramite factory → **`valueOf`**

```
Integer i = Integer.valueOf(22);
Double d  = Double.valueOf(3.14);
```

Java

    - ottimizza la creazione delle istanze, mediante caching
    - *value-based classes:* «object identity should not matter»

VERSO VALUE-BASED CLASSES

- Il concetto di *value-based class* è già sfruttato in Java per `Optional` e le classi del package `java.time`
- Le istanze di una value-based class:
  - sono *final* e *immutabili* (possono però referenziare oggetti mutevoli)
  - utilizzano solo equals per l'uguaglianza: non usano mai operazioni sensibili all'identità (come == o hashcode)
  - non espongono costruttori: sono istanziate solo da metodi factory, per non dare/fare ipotesi sulla *identità* degli oggetti restituiti
  - `equals`, `hashCode`, `toString` sono basate solo sullo *stato* dell'istanza stessa (non sulla sua identità o sullo stato di altri oggetti)
  - perciò, due istanze uguali secondo `equals` *sono sempre liberamente intercambiabili* senza che ciò causi modifiche di comportamento

VERSO VALUE-BASED CLASSES

- Con i vecchi costruttori espliciti, venivano create istanze distinte anche per lo stesso valore:

```
jshell> Integer i = new Integer(18)
i ==> 18

jshell> Integer j = new Integer(18)
j ==> 18

jshell> i==j
$3 ==> false
```

Java

- Con i nuovi factory method, ciò non accade:

```
jshell> Integer u = Integer.valueOf(17)
u ==> 17

jshell> Integer v = Integer.valueOf(17)
v ==> 17

jshell> u==v
$6 ==> true
```

Java

CLASSI WRAPPER IN JAVA: BOXING & UNBOXING

- Analogamente, l'estrazione del valore primitivo dal wrapper si chiama *UNBOXING* ed è svolta dai metodi **`xxxValue`**

Modifier and Type	Method	Description
byte	**byteValue**()	Returns the value of the specified number as a byte.
abstract double	**doubleValue**()	Returns the value of the specified number as a double.
abstract float	**floatValue**()	Returns the value of the specified number as a float.
abstract int	**intValue**()	Returns the value of the specified number as an int.
abstract long	**longValue**()	Returns the value of the specified number as a long.
short	**shortValue**()	Returns the value of the specified number as a short.

Java

  - di norma, si usa il metodo di estrazione «corrispondente» al wrapper

```
int x    = i.intValue();      // x vale 22
double z = d.doubleValue();   // z vale 3.14
```

  - tuttavia, sono sempre disponibili tutti i metodi, che restituiscono il valore *convertito* nel tipo richiesto

CLASSI WRAPPER IN JAVA: BOXING & UNBOXING

- Esempi d'uso

Java

```
Integer i = Integer.valueOf(22);
System.out.println(i.intValue());       // 22
System.out.println(i.doubleValue());    // 22.0

Double d = Double.valueOf(3.14);
System.out.println(d.doubleValue());    // 3.14
System.out.println(d.intValue());       // 3

Float f = Float.valueOf(1.73F);
System.out.println(f.floatValue());     // 1.73
System.out.println(f.doubleValue());    // 1.730000019…
System.out.println(f.longValue());      // 1
```

BOXING & UNBOXING AUTOMATICI

- L'uso esplicito di boxing & unboxing renderebbe il codice molto (inutilmente) prolisso:
  - sia nelle operazioni

```
Integer k = Integer.valueOf( i.intValue()+6 );
```

  - sia nell'uso con le collection

```
myList.add( Integer.valueOf(7) );
```

Java

- Per questo, da Java 5 boxing e unboxing sono *automatici*
  - gli automatismi garantiscono l'approccio *value-based*

```
jshell> Integer k = 16
k ==> 16

jshell> Integer h = 16
h ==> 16

jshell> k==h
$9 ==> true
```

```
jshell> int z = k
z ==> 16

jshell> int x = k-1+3*h
x ==> 63
```

BOXING & UNBOXING NELLE COLLECTION

Java

```java
public static void main(String[] args){

    java.util.List list = new java.util.ArrayList();
    list.add(23); // boxing: int --> Integer
    list.add(24); // boxing: int --> Integer
    list.add(25); // boxing: int --> Integer
    for(Object obj : list) System.out.println(obj);
    int u = (int) list.get(0);  // unboxing: Integer --> int
    int v = (int) list.get(2);  // unboxing: Integer --> int
    System.out.println(u);
    System.out.println(v);

    java.util.List<Integer> listOfInt = new java.util.ArrayList<>();
    listOfInt.add(23); // boxing: int --> Integer
    listOfInt.add(24); // boxing: int --> Integer
    listOfInt.add(25); // boxing: int --> Integer
    int x = listOfInt.get(0); // boxing: int --> Integer
    int y = listOfInt.get(1); // boxing: int --> Integer
    int z = listOfInt.get(2); // boxing: int --> Integer
    System.out.println(x);
    System.out.println(y);
    System.out.println(z);
}
```

```
23
24
25
23
25
```

```
23
24
25
```

C#, Scala, Kotlin: NO BOXING !

C#

```
public static void Main()
{
    System.Collections.IList list = new System.Collections.ArrayList();
    list.Add(23);
    list.Add(24);
    list.Add(25);
    foreach(Object obj in list) Console.WriteLine(obj);
    int u = (int) list[0]; Console.WriteLine(u);
    int v = (int) list[2]; Console.WriteLine(v);

    System.Collections.Generic.IList<int> intlist = new System.Collections.Generic.List<int>();
    intlist.Add(23);
    intlist.Add(24);
    intlist.Add(25);
    int x = intlist[0]; Console.WriteLine(x);
    int y = intlist[1]; Console.WriteLine(y);
    int z = intlist[2]; Console.WriteLine(z);
```

In C#, int è una vera classe (alias per System.Int32)

Inoltre, List implementa IList

Kotlin

```
fun main() {

    val list : kotlin.collections.MutableList<Int> = kotlin.collections.ArrayList();
    list.add(23);
    list.add(24);
    list.add(25);
    for(i in list) println(i);
    val x = list.get(0); println(x);
    val y = list.get(1); println(y);
    val z = list.get(2); println(z);
}
```

In Kotlin, ArrayList è un'implementazione di MutableList

Int è una vera classe

Scala

```
def main(args: Array[String]) : Unit = {

  val list : scala.collection.mutable.ArrayDeque[Int] = scala.collection.mutable.ArrayDeque();
  list += 23;
  list += 24;
  list += 25;
  for(i ← list) println(i);
  val x = list(0); println(x);
  val y = list(1); println(y);
  val z = list(2); println(z);
```

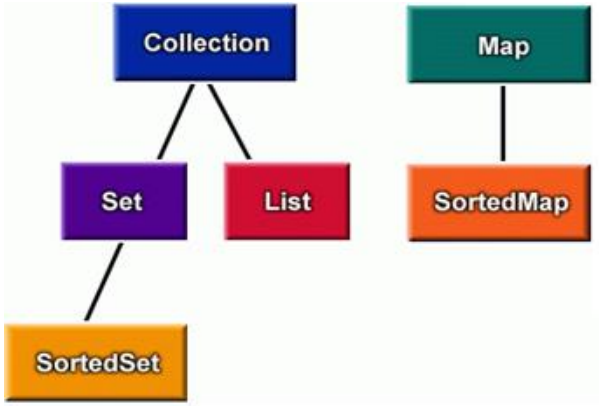
Int è una vera classe

In Scala 2.13, ArrayDeque è un'implementazione molto efficiente di lista modificabile

Java Collection Framework

JAVA COLLECTION FRAMEWORK (JCF) (package `java.util`)

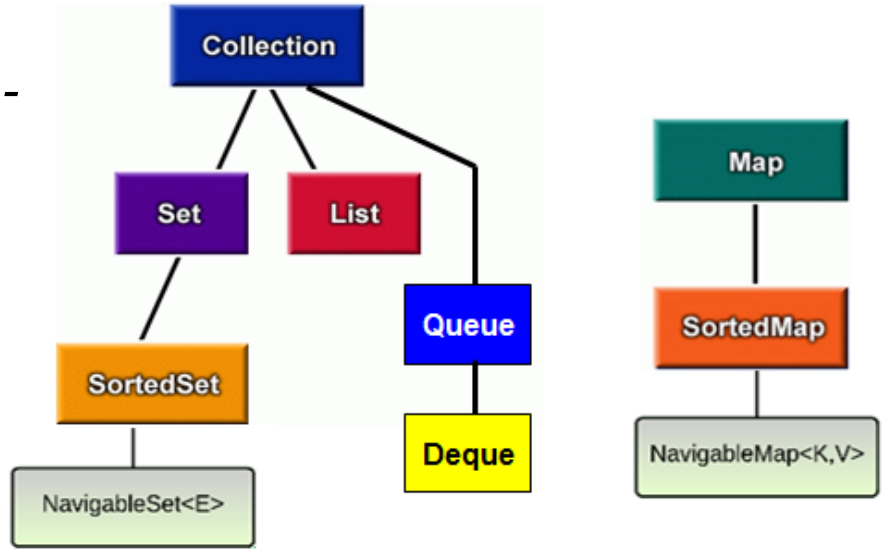
- In Java, le interfacce-chiave sono quattro:
  - **`Collection<T>`**
    nessuna ipotesi specifica
  - **`Set<T>`** (variante: **`SortedSet<T>`**)
    insieme di elementi (quindi, senza duplicati)
  - **`List<T>`**
    sequenza indicizzata di elementi
  - **`Map<K,V>`** (variante: **`SortedMap<K,V>`**)
    tabella che associa chiavi a valori



- In che senso alcune sono «*Sorted*»?
  - l'aggettivo si riferisce alla *navigabilità*, ossia al modo con cui si esplora la collezione (es. nel *foreach*)
  - la lista ovviamente ha già un suo criterio intrinseco di «sequenza», legato alla sua natura, che però non implica «ordinamento»

JAVA COLLECTION FRAMEWORK (JCF) (package `java.util`)

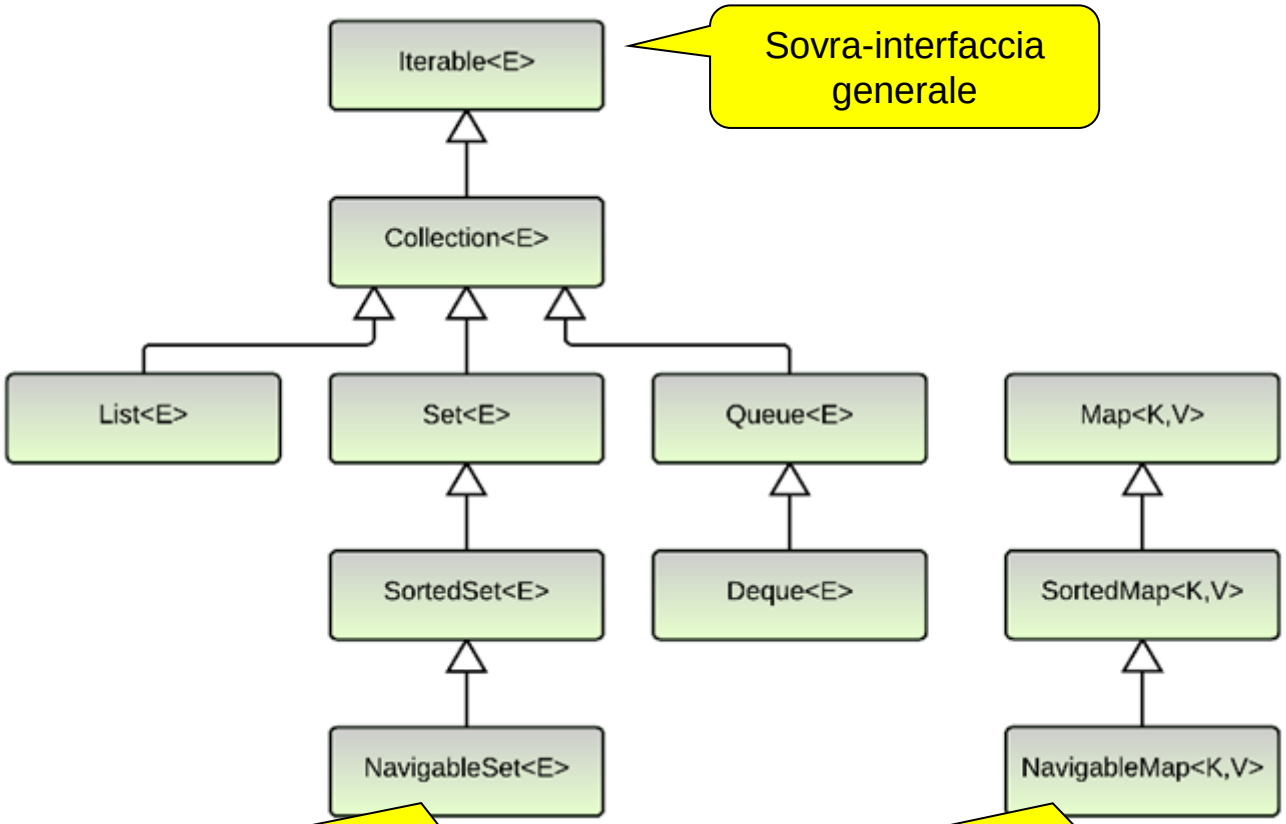
- Alle interfacce fondamentali si aggiungono:
  - **`Queue<T>`**
    coda di elementi (non necessaria-mente FIFO: sono "code" anche gli stack, che operano LIFO)
  - **`Deque<T>`**
    doppia coda (coda circolare), *doppiamente concatenata*



- In tempi più recenti anche:
  - **`NavigableSet<T>`**
    specializzazione di SortedSet
  - **`NavigableMap<K,V>`**
    specializzazione di SortedMap

  entrambe navigabili sia in senso ascendente che discendente

JCF: QUADRO GENERALE INTERFACCE



JCF: SCELTE DI FONDO

Java

- Le interfacce sono progettate secondo due criteri:
  - Minimalità: prevedere solo metodi davvero basilari…
  - Efficienza: …o che migliorino nettamente le prestazioni
- Le classi che le implementano si ispirano a
  - Scelta: offrire una o due scelte per ogni interfaccia..
  - Efficienza: …adatte per prestazioni a situazioni diverse
- Factory solo per creare *costanti literal immodificabili*
  - Ufficialmente: si vuole che l'utente scelga l'implementazione
  - In realtà: vero, ma JCF è anche figlia del suo tempo
- Libreria **`Collections`**
  - Contiene svariati metodi statici per operare sulle collezioni

L'INTERFACCIA `Iterable<T>`

- **`Iterable<T>`** introduce l'idea di *qualcosa di iterabile* Java
  - interfaccia di top-level introdotta in Java 5 insieme al *foreach*
  - *"Implementing this interface allows an object to be the target of the for-each loop statement"*
- Risponde alla necessità di poter «iterare» (ciclare) su strutture dati *arbitrarie* in modo *generale e flessibile*
  - gli array hanno gli indici, ma molte altre strutture no
  - **`Iterable`** è la chiave per poter supportare il costrutto *for each su qualunque struttura dati presente e futura*, tramite *iteratori*
  - se un domani volessimo realizzare un nuovo tipo di struttura dati, basterà renderla «iterabile» perché sia riconoscibile dal *for each*

Design for the future

L'INTERFACCIA `Collection<T>`

- **`Collection<T>`** introduce l'idea di *collezione di elementi*
  - non si fanno ipotesi sulla natura di tale collezione, tranne il fatto che sia «iterabile» → *ci si potrà navigare dentro con il for each* ☺
  - in particolare, non si dice che sia un insieme o una sequenza, che ci sia o meno un ordinamento, che ammetta duplicati, etc.
- L'interfaccia di accesso è *volutamente generale*: Java
  - assicurarsi che un elemento sia nella collezione **`add`**
  - rimuovere un elemento dalla collezione **`remove`**
  - verificare se un elemento è nella collezione **`contains`**
  - verificare se la collezione è vuota **`isEmpty`**
  - sapere la cardinalità della collezione **`size`**
  - ottenere un array con gli stessi elementi **`toArray`**
  - verificare se due collezioni sono "uguali" **`equals`**
  - *… e altri …*

L'INTERFACCIA `Collection<T>`

- **`Collection<T>`** introduce l'idea di *collezione di elementi*
  - non si fanno ipotesi sulla natura di tale collezione, tranne il fatto che sia «iterabile» → *ci si potrà navigare dentro con il for each* ☺
  - in particolare, non si dice che sia un insieme o una sequenza, che ci sia o meno un ordinamento, che ammetta duplicati, etc.
- L'in*erale*:
  - **`add`**
  - **`remove`**
  - **`contains`**
  - **`isEmpty`**
  - **`size`**
  - **`toArray`**
  - verificare se due collezioni sono "uguali" — **`equals`**
  - *… e altri …*

OSSERVA: non esiste un metodo `get`

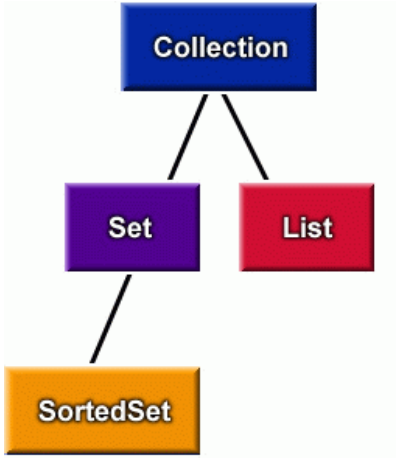
Non è casuale: non c'è modo di recuperare un singolo elemento in modo specifico perché non esiste un *criterio* per farlo.

Si può solo *iterare* sulla collezione, *verificare la presenza* di un elemento, o eventualmente *rimuoverlo*.

Java

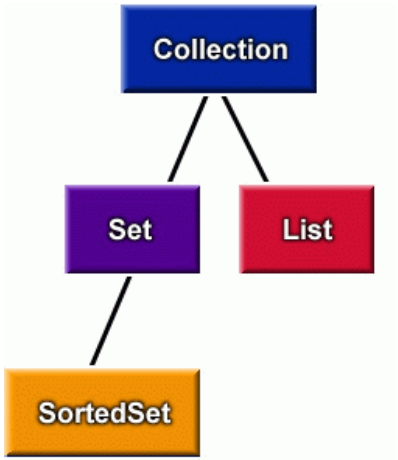
L'INTERFACCIA `Set<T>`

- **`Set<T>`** specializza **`Collection<T>`** introducendo l'idea di *insieme di elementi (privo di duplicati)*
  - in quanto insieme, non ha una nozione di sequenza o di posizione
- L'interfaccia di accesso non cambia sintatticamente, ma prevede nuovi vincoli al contratto d'uso:
  - **`add`** aggiunge un elemento di tipo **`T`** solo se esso non è già presente
  - **`equals`** assicura che due **`Set`** siano identici nel senso insiemistico ($\forall x \in S1, x \in S2$ e viceversa)
  - tutti i costruttori si impegnano a creare *insiemi privi di duplicati*



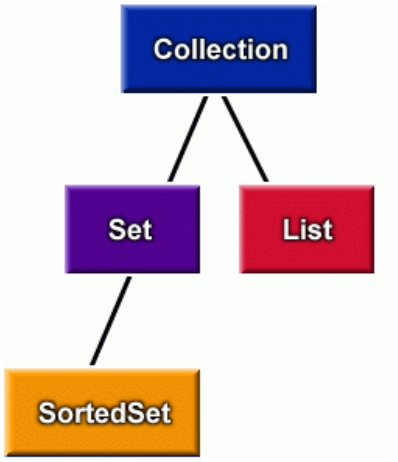
L'INTERFACCIA `SortedSet<T>`

- **`SortedSet<T>`** specializza **`Set<T>`** aggiungendo la nozione di *ordinamento totale fra gli elementi*
  - gli elementi devono essere **`Comparable<T>`**, altrimenti occorre fornire un **`Comparator<T>`** al momento della costruzione
  - l'iteratore naviga nella struttura *seguendo l'ordine*
- L'interfaccia di accesso aggiunge metodi che sfruttano l'ordinamento totale fra gli elementi:
  - **`first`** e **`last`** restituiscono il primo e l'ultimo elemento (di tipo **`T`**)
  - **`headSet`**, **`subSet`** e **`tailSet`** restituiscono i *sottoinsiemi ordinati* contenenti rispettivamente i soli elementi *minori* di quello dato, *compresi* fra due elementi dati, o *maggiori* dell'elemento dato.



L'INTERFACCIA `List<T>`

- **`List<T>`** specializza **`Collection<T>`** introducendo l'idea di *lista di elementi*
  - molto simile a un array (esistono metodi di conversione)
  - ha una nozione di sequenza, di posizione e ammette duplicati
  - l'iteratore naviga nella struttura *seguendo la sequenza*
- L'interfaccia aggiunge vincoli al contratto d'uso e definisce nuovi metodi per l'accesso posizionale:
  - **`add`** aggiunge un elemento in fondo alla lista (semantica di append)
  - **`equals`** è vero se gli elementi corrispondono a due a due (o sono entrambi `null`)
  - nuovi metodi **`add`**, **`remove`**, **`get`** che accedono alla lista anche *per posizione*

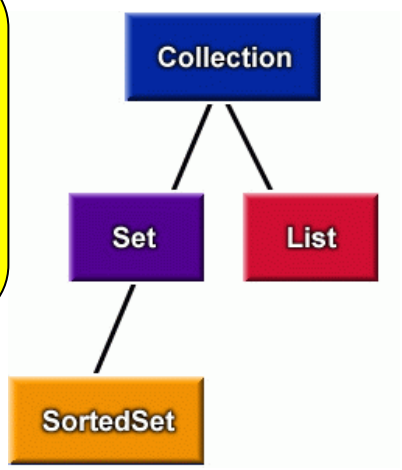


L'INTERFACCIA `List<T>`

- **`List<T>`** specializza **`Collection<T>`** introducendo l'idea di *lista di elementi*
  - molto simile a un array (esistono metodi di conversione)
  - ha una nozione di sequenza, di posizione e ammette duplicati
  - l'iteratore naviga nella struttura *seguendo la sequenza*
- L'interfaccia aggiunge vincoli al contratto d'uso e definisce n
  - nuovi metodi **`add`**, **`remove`**, **`get`** che accedono alla lista anche *per posizione*

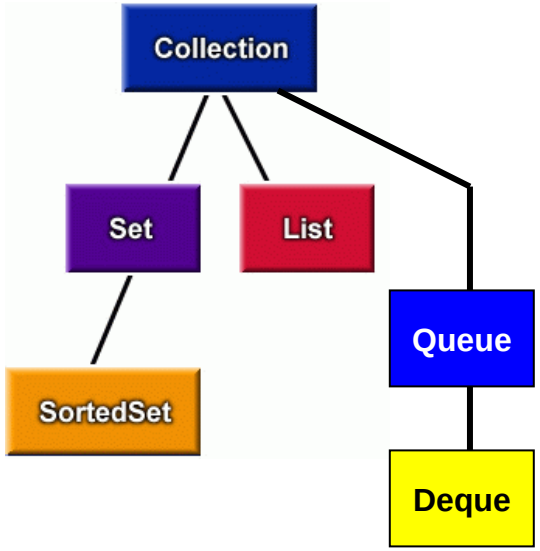
OSSERVA: qui compare il metodo `get`

Non è casuale: la nozione di *posizione* introduce esattamente il *criterio* che permette di farlo: con un *indice*!



INTERFACCE `Queue<T>` & `Deque<T>`

- **`Queue<T>`** specializza **`Collection<T>`** introducendo la nozione di *coda di elementi*
  - ha la nozione di *testa della coda,* ma *non* di posizione (indice)
- **`Deque<T>`** la specializza con l'idea di *doppia coda*
  - si possono inserire/togliere elementi da entrambe le estremità
- L'interfaccia di accesso si specializza:
  - **`remove`** estrae e rimuove l'elemento *in testa*
  - **`element`** estrae l'elemento *in testa* alla coda senza rimuoverlo
  - analoghi metodi, con nome simile, restituiscono invece una *indicazione di fallimento* anziché lanciare eccezione

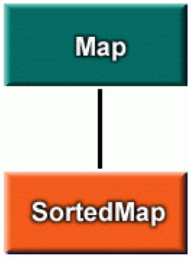


L'INTERFACCIA `Map<K,V>`

- **`Map<K,V>`** non è una Collection perché è una struttura bidimensionale, una *tabella* di elementi (*valori*) associati ciascuno a una *chiave* identificativa univoca

chiave	*valore*
key1	oggetto1
key2	oggetto2
…	…

  - è una tabella a due colonne (chiavi, elementi) in cui *i dati della prima colonna (chiavi) identificano univocamente la riga*
  - si può così accedere agli elementi *per chiave,* idealmente in un tempo costante, superando l'accesso sequenziale "stile lista"

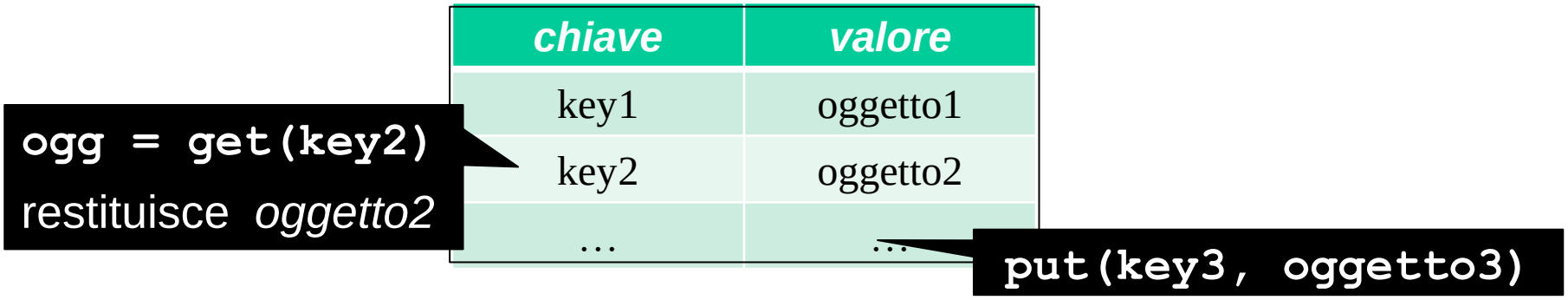


  - lo si ottiene sfruttando *funzioni matematiche (hash)* che mettono in corrispondenza *chiavi* e *valori:* data la chiave, la funzione restituisce *la posizione d*ell'elemento
  - in alternativa, si possono predisporre *indici* che guidino il reperimento dell'elemento a partire dalla chiave.

L'INTERFACCIA `Map<K,V>`

Java

- L'interfaccia di accesso prevede metodi per:
  - *inserire* in tabella una coppia (chiave, elemento) **`put`**
  - *accedere* a un elemento in tabella, data la chiave **`get`**
  - verificare se una *chiave* è presente in tabella **`containsKey`**
  - verificare se un *elemento* è presente **`containsValue`**
- In particolare:
  - **`put`** inserisce in tabella una nuova riga (nella riga "opportuna")
  - **`get`** recupera *a colpo sicuro* un elemento



L'INTERFACCIA `Map<K,V>`

Java

- L'interf
  - *ins* `put`
  - *ac* `get`
  - ver `containsKey`
  - ver `ontainsValue`
- In parti
  - **`put`** inseris a una nuova riga (nella riga "opportuna")
  - **`get`** recupera *a colpo sicuro* un elemento

OSSERVA: non esiste un «indice di riga»

Non è casuale: la tabella va vista come insieme di righe *indipendenti*.

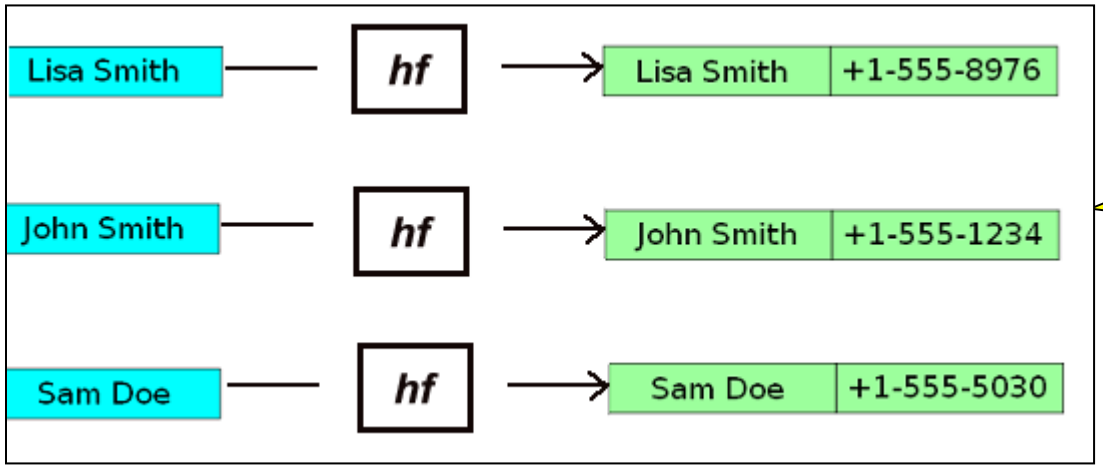
Il criterio di accesso deve essere basato sulla chiave, non su una «posizione» indefinita e priva di significato (non è Excel!)

chiave	*valore*
key1	oggetto1
key2	oggetto2
…	…

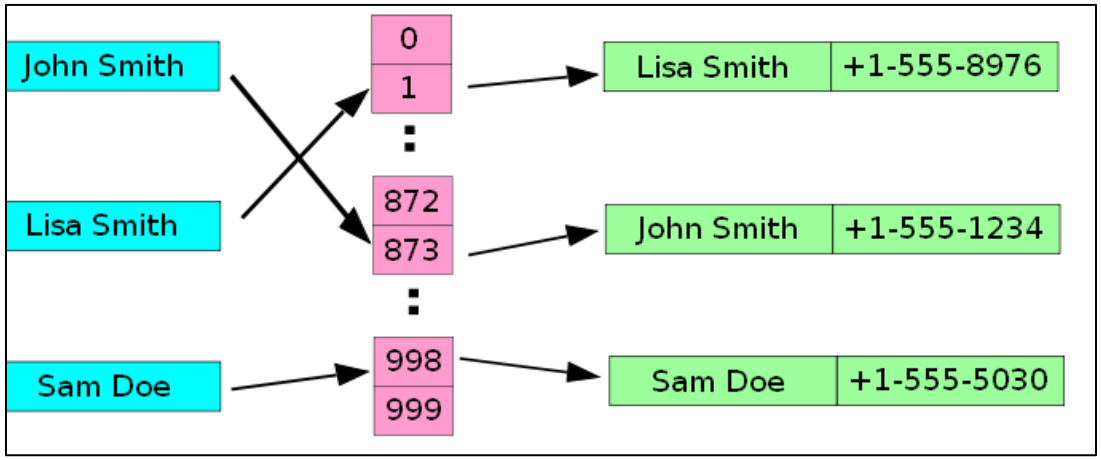
`ogg = get(key2)`
restituisce *oggetto2*

`put(key3, oggetto3)`

IL CONCETTO DI MAPPA



Mappe basate su *funzioni hash*



Mappe basate su *indici*

L'INTERFACCIA `Map<K,V>` & il metodo `hashcode`

- Perché la mappa funzioni è essenziale che la funzione hash produca *valori distinti per elementi distinti*
- A tal fine, la classe **`Object`** definisce il metodo **`hashcode`**
  - ereditato da tutte le classi, specializzato da molte di esse
- Regola aurea: se si ridefinisce **`equals`** va ridefinito anche **`hashcode`** *in modo coerente*
  - tutte le classi standard lo fanno: *per le nostre dovremo pensarci noi* altrimenti get, put, assertEquals si comporteranno in modo inatteso
  - in generale, occorre combinare gli hashcode dei campi-dati secondo certe regole, usando come coefficienti dei *numeri primi*
  - ma per fortuna non occorre sapere come fare: Eclipse (o altro tool) può generarne una versione "standard"

L'INTERFACCIA `Map<K,V>`
Il metodo `hashcode`

- Esempio di **`hashcode`** generato automaticamente per una banale classe con due campi stringa (nome e email):

Java

```
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + ((email==null) ? 0 : email.hashCode());
  result = prime * result + (int) (id ^ (id >>> 32));
  result = prime * result + ((name==null) ? 0 : name.hashCode());
  return result;
}
```

L'INTERFACCIA `Map<K,V>`
Il metodo `hashcode`

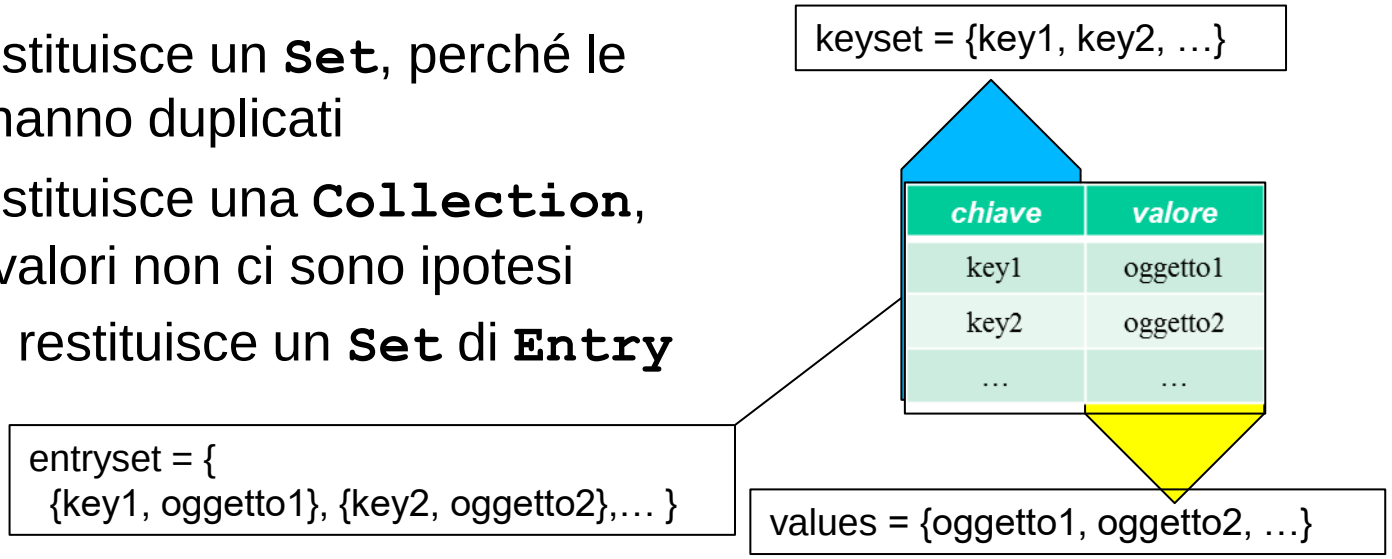
- Esempio di **`hashcode`** generato automaticamente per una classe con più campi, di vario tipo:
  - `float altezza, int peso`
  - `String citta, LocalDate data`

Java

```
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + Float.floatToIntBits(altezza);
  result = prime * result +
                   ((citta == null) ? 0 : citta.hashCode());
  result = prime * result +
                   ((data == null) ? 0 : data.hashCode());
  result = prime * result + peso;
  return result;
}
```

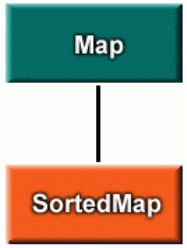
L'INTERFACCIA `Map<K,V>` Collection views

- Poiché **`Map`** non è una collection, appositi metodi permettono di estrarre i dati sotto forma di *Collection views*:
  - tutta la colonna *chiavi* (K) **`keySet`**
  - tutta la colonna *valori* (V) **`values`**
  - tutte le *righe* <K,V> (coppie chiave, elemento) **`entrySet`**
- In particolare:
  - **`keyset`** restituisce un **`Set`**, perché le chiavi non hanno duplicati
  - **`values`** restituisce una **`Collection`**, perché sui valori non ci sono ipotesi
  - **`entryset`** restituisce un **`Set`** di **`Entry`** (le righe)



L'INTERFACCIA `SortedMap<K,V>`

- **`SortedMap<K,V>`** specializza **`Map<K,V>`** nello stesso modo in cui `SortedSet<T>` specializza `Set<T>`
  - le chiavi devono essere `Comparable<K>`
  - in alternativa, occorre fornire un `Comparator<K>` in fase di creazione della mappa
  - l'iteratore naviga nella mappa seguendo *l'ordine delle chiavi*



- L'interfaccia di accesso aggiunge metodi che sfruttano l'ordinamento totale fra le chiavi:
  - **`firstKey`** e **`lastKey`** restituiscono la *prima* e *l'ultima* chiave
  - **`headMap`**, **`subMap`** e **`tailMap`** restituiscono le *sottomappe ordinate* contenenti le sole righe le cui chiavi siano rispettivamente *minori* di quella data, *comprese* fra due chiavi, o *maggiori* della chiave data.

LA LIBRERIA `Collections`

- In Java, la Libreria **`Collections`** contiene *funzioni statiche* per operare sulle collezioni
  - algoritmi polimorfi che operano su qualunque tipo di collezione per *ordinamento*, *ricerca binaria*, *riempimento*, *ricerca min/max…*
    - **`sort(List<T> list)`**: ordina una lista con un merge sort ottimizzato, che garantisce tempi dell'ordine di n*log(n)
    - **`binarySearch(List<T> list,T element)`**
  - *wrapper* per incapsulare una collezione di un tipo in un'altra
  - nonché tre *costanti* di utilità
    - la lista vuota **`EMPTY_LIST`**
    - l'insieme vuoto **`EMPTY_SET`**
    - la mappa vuota **`EMPTY_MAP`**

In linguaggi successivi tali algoritmi sono solitamente definiti nelle interfacce/tratti stesse

LA LIBRERIA `Collections`

- In Java, la Libreria **`Collections`** contiene *funzioni statiche* per operare sulle collezioni
  - algoritmi polimorfi che operano su qualunque tipo di collezione per *ordinamento*, *ricerca binaria*, *riempimento*, *ricerca min/max…*
    - **`sort(List<T> list)`**: ordina una lista con un merge sort ottimizzato, che garantisce tempi dell'ordine di n*log(n)
    - **`binarySearch(List<T> list,T element)`**
  - *wr* … in un'altra
  - nc…
    - …*guaggi successivi*
    - …*goritmi sono solita-*
    - …*e definiti nelle*
    - …*acce/tratti stesse*

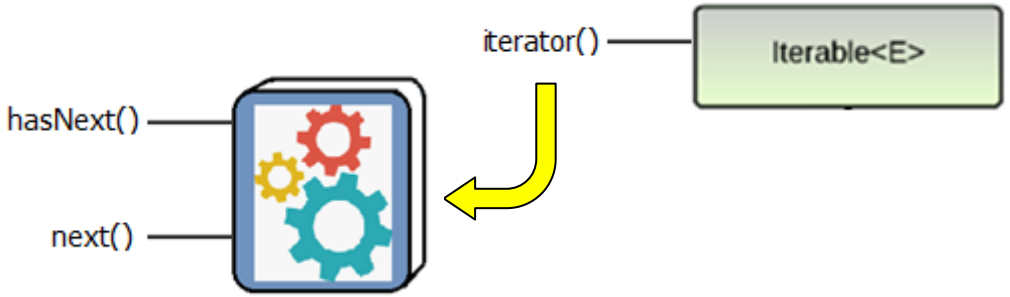
OSSERVA: non si può ordinare una collection che non sia una lista!

MOTIVO: «ordinare» implica cambiare *posizione* agli elementi, quindi richiede per ciò stesso una qualche nozione di *posizione*

Idem per la ricerca binaria

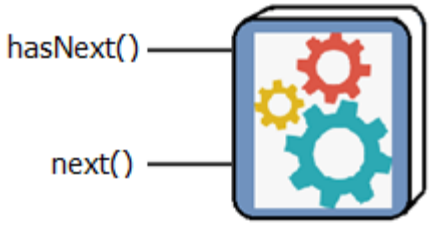
RIPRENDIAMO L'INTERFACCIA `Iterable<T>`

- Esprime l'idea di un'entità «iterabile», ossia su cui è possibile ciclare tramite *un for-each*
- Come funziona?
  - IDEA BASE: ogni cosa iterabile «sa» come navigare in sé stessa
  - quindi, a richiesta, è in grado di produrre un *componente in grado di restituire via via tutti gli elementi* → un *iteratore*
  - il metodo **`iterator()`** è appunto una mini-factory *disponibile per ipotesi in ogni cosa «iterabile»* che produce l'iteratore adatto a navigare in quel certo tipo di struttura



UN NUOVO CONCETTO: ITERATORI

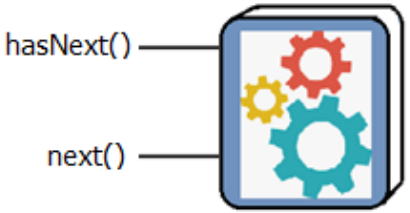
- Perché gli iteratori?
  - negli array si naviga tramite indici, ma in una collezione qualsiasi non esiste proprio una nozione di «sequenza», quindi…
  - serve un mezzo per navigare su una collezione di elementi *senza fare ipotesi* sull'esistenza di indici, posizioni, puntatori.. e amenità varie
- L'*iteratore* è appunto, per definizione, un oggetto capace di *navigare* in una cosa «iterabile» del «suo» tipo
  - estrae da essa un elemento per volta
  - garantisce che ogni elemento venga considerato *una e una sola volta*, indipendentemente
    - dal tipo di collezione
    - da come è realizzata



NB: l'iteratore *non si applica* alle mappe, che sono bidimensionali e infatti non derivano da Collection

ITERATORI

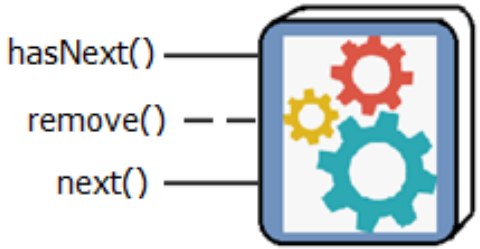
- Un iteratore per una **`Collection<T>`** è un oggetto che implementa l'interfaccia **`Iterator<T>`**
  - una collezione di **`Frazione`** richiede un **`Iterator<Frazione>`**
  - una lista di **`PhonePlan`** richiede un **`Iterator<PhonePlan>`**
  - etc.
- Di norma, *non* occorre manipolarlo direttamente
  - si usa *indirettamente*, tramite il costrutto *for each*
- Però, è possibile anche gestirlo esplicitamente
  - lo si chiede alla collection → metodo **`iterator`**
  - lo si fa avanzare invocando il metodo **`next`** …
  - … fino a quando il metodo **`hasNext`** ritorna *false*



L'INTERFACCIA `Iterator`

Java

- L'interfaccia **`Iterator<T>`** dichiara tre metodi:
  - **`next`** che restituisce "il prossimo" elemento
  - **`hasNext`** che dice se ci sono o no ulteriori elementi
  - **`remove`** (opzionale) che rimuove l'ultimo elemento restituito
    - le collezioni NON sono obbligate a implementarlo
    - tuttavia, quelle della JCF standard lo fanno
- Perché un metodo «opzionale»?
  - per non costringere a implementarlo anche dove non avrebbe senso → entità *immutabili*



- Cosa fa chi non lo implementa?
  - lo implementa «per finta», lanciando l'errore (eccezione) *UnsupportedOperationException*

ITERATORI NEL FOR-EACH

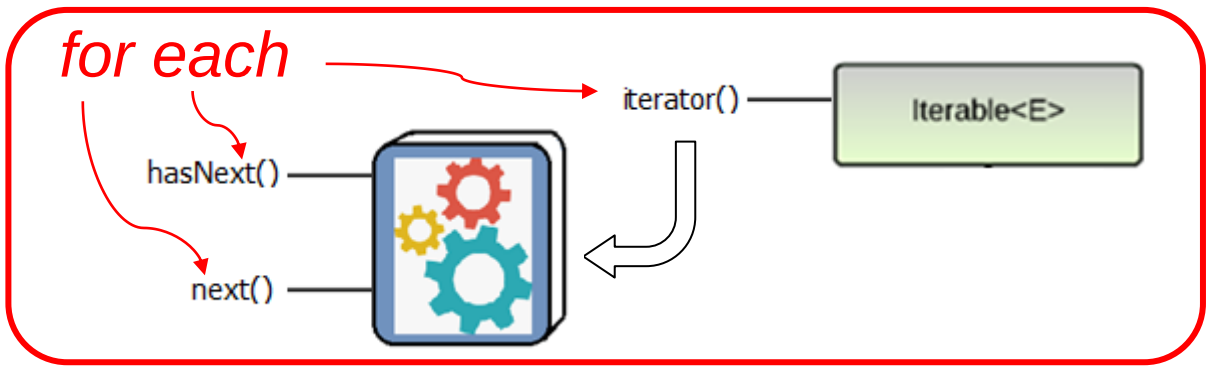
- L'iteratore è alla base del costrutto *for each*
  - in *qualunque entità «iterabile»* (inclusi gli array):

Java

```
for(tipo x : coll){ /* operazioni su x */ }
```

  - il costrutto *for each* sopra equivale a:

```
for( Iterator<tipo> i = coll.iterator();
     i.hasNext() ){
    /* operazioni su x = i.next() */
}
```



ITERATORI: ESEMPIO OK

Java

```java
import java.util.*;

class IteratorExample {

    public static void main(String[] args) {

        List<String> listOfStrings = List.of("Pippo", "Pluto", "Paperino", "Zio Paperone");
        iterateOn("lista disney", listOfStrings);
        Set<Number> setOfNums = Set.of(18,22.2, 37.4F);
        iterateOn("numeri vari", setOfNums);

    }

    public static <T> void iterateOn(String msg, Collection<T> coll) {
        System.out.println("--------------------");
        System.out.println(msg);

        System.out.println("uso di iteratore implicito");
        for (T s : coll) System.out.println(s);

        System.out.println("uso di iteratore esplicito");
        Iterator<T> it = coll.iterator();
        while (it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

```
--------------------
lista disney
uso di iteratore implicito
Pippo
Pluto
Paperino
Zio Paperone
uso di iteratore esplicito
Pippo
Pluto
Paperino
Zio Paperone
--------------------
numeri vari
uso di iteratore implicito
18
22.2
37.4
uso di iteratore esplicito
18
22.2
37.4
```

Due collezioni *diverse* con elementi di tipi *totalmente diversi*: yet, it works!

Funzione generica: opera su elementi di tipo T, *qualunque cosa sia T*

Opera su elementi di tipo T

Iteratore (per elementi di tipo T) usato in modo esplicito su una collezione *del suo tipo* T

ITERATORI: COSA SÌ, COSA NO

- Cosa si può fare con un iteratore?
  - iterare su una collezione per accedere agli elementi
  - iterare su una collezione *modificabile* per modificare gli elementi
- Cosa NON si può fare con un iteratore?
  - modificare la collezione stessa su cui si sta iterando (cioè, aggiungere o togliere elementi mentre l'iteratore sta lavorando)
  - se si viola questo vincolo, **`ConcurrentModificationException`**
- Ciò va tenuto presente quando si opera col *for each*
  - OK lavorare sugli elementi che via via si scorrono
  - NO aggiungere/togliere elementi dalla collezione su cui si sta iterando (è come segare il ramo dell'albero su cui si è seduti..)

ITERATORI: ESEMPIO KO

Java

```java
import java.util.*;

class BadIteratorExample {

    public static void main(String[] args) {

        List<String> listOfStrings = List.of("Pippo", "Pluto", "Paperino", "Zio Paperone");
        iterateAndChange("lista disney", listOfStrings); // UnsupportedOperationException: ImmutableCollections

        List<Number> listOfNums = new ArrayList<Number>();
        listOfNums.add(21); listOfNums.add(2.4); listOfNums.add(-1.3F);
        iterateAndChange("numeri vari", listOfNums);     // ConcurrentModificationException

    }

    public static <T> void iterateAndChange(String msg, Collection<T> coll) {
        System.out.println("--------------------");
        System.out.println(msg);

        System.out.println("uso di iteratore implicito");
        for (T element : coll) {
            System.out.println(element);
            coll.add(element); // ARGH!! Aggiunge un duplicato!!
        }

    }
}
```

Tentativo di modificare la collezione *mentre ci si sta iterando sopra*

- il primo caso fallisce perché la lista prodotta dalla factory **`List.of`** è immodificabile
- il secondo caso rende inconsistente lo stato dell'iteratore, a cui cede letteralmente «il terreno sotto i piedi» perché gli si altera «in fieri» la collezione su cui sta lavorando

ITERATORI & MAPPE

- L'iteratore naviga *collezioni di elementi*, ossia contenitori *monodimensionali*
  - infatti, è una generalizzazione del concetto di "variabile di ciclo"
- Come tale, *non si può usare direttamente sulle mappe,* che sono invece contenitori bidimensionali
  - d'altronde, cosa vorrebbe dire "iterare" su una mappa? a zig-zag..?
- Si può però usare sulle *singole colonne* della mappa
  - ottenibili rispettivamente tramite i metodi **`keySet()`** e **`values()`**
- oppure sull'*insieme di righe* della mappa (set di **`Map.Entry`**)
  - ottenibile tramite il metodo **`entrySet()`**

È una ***inner class*** (sorta di "classe interna" di **Map**)

ITERATORI & MAPPE

Java

```
import java.util.*;

class MapIteratorExample {

    public static void main(String[] args) {

        Map<String,Integer> peopleMap = Map.of(
            "Anna", 21, "Piero", 25, "Silvia", 43, "Guido", 56 );

        // collection views: keySet
        for(String name : peopleMap.keySet()){
            System.out.println(name + " ha " + peopleMap.get(name) + " anni" );
        }
        System.out.println();

        // collection views: values
        float sum=0;
        for(int value: peopleMap.values()){
            sum +=value;
        }
        System.out.println("L'età media è di " + sum/peopleMap.size() + " anni" );
        System.out.println();

        // collection views: entries
        for(Map.Entry entry: peopleMap.entrySet()){
            System.out.println(entry);
        }

    }
```

```
Silvia ha 43 anni
Piero ha 25 anni
Anna ha 21 anni
Guido ha 56 anni

L'età media è di 36.25 anni

Silvia=43
Piero=25
Anna=21
Guido=56
```

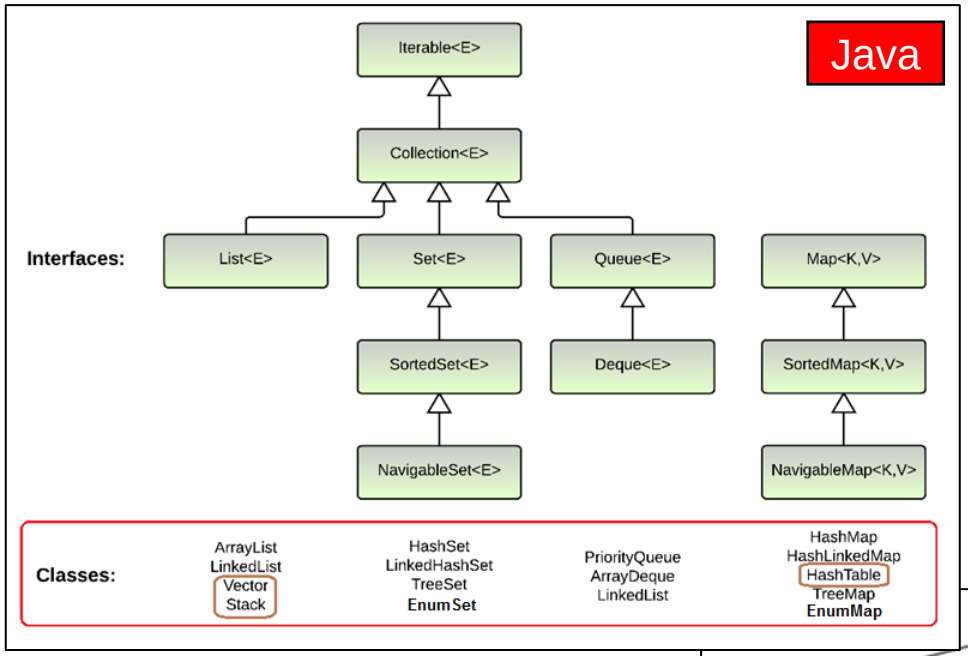
Recupera il valore associato alla chiave corrente

Divide la somma per la dimensione (#righe) della tabella

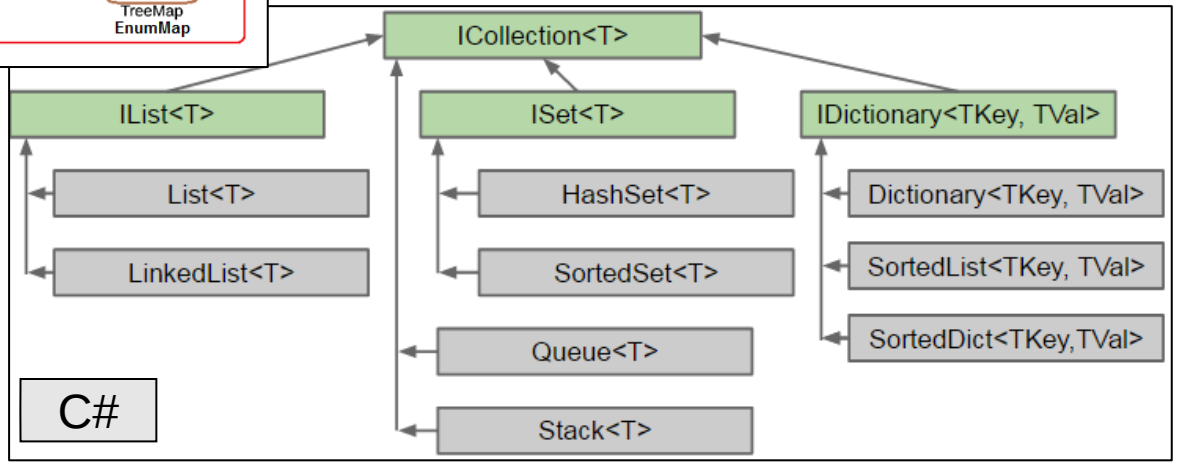
La toString di Entry stampa una frase della forma "nome=valore"

Generalizzazione
Collection Framework in altri linguaggi

COLLECTION FRAMEWORK: UN ESEMPIO PLURI-LINGUAGGIO



Implementazioni modificabili



ESEMPIO PLURI-LINGUAGGIO: Set in Java

- L'interfaccia **`Set`** definisce un *insieme*
- La factory internalizzata **`Set.of`** produce un set *immodificabile*
- Le altre implementazioni di `Set`, `HashSet` e `TreeSet`, producono set *modificabili.* In particolare, `TreeSet` dà luogo a un insieme *ordinato*, perché implementa `SortedSet`.
- L'iteratore implicito del *foreach* naviga il set in modo arbitrario: garantisce invece di seguire l'ordine nel caso di `SortedSet` (e quindi di `TreeSet`)
- Notare il costruttore per copia: tutte le collection (qui `TreeSet` in `s3`) accettano come argomento un'altra collection (qui, un `Set`) di cui copiano in modo shallow tutti gli elementi.

Java

```
public static void main(String []args){
   Set<Double> s1 = Set.of(3.14, 2.15, 1.16, -0.17);
   Set<Double> s2 = new HashSet<>();
   s2.add(-3.14); s2.add(-2.15);
   s2.add(-1.16); s2.add(0.17);
   System.out.println(s1);
   System.out.println(s2);
   // s1.add(22.2); // UnsupportedOperationException
   s2.add(11.1);
   System.out.println(s2);
   for (double d : s1) System.out.println("Elemento: " + d);
   Set<Double> s3 = new TreeSet<>(s1);
   s3.add(22.2);
   System.out.println(s3);
   for (double d : s3) System.out.println("Elemento: " + d);
}
```

```
[2.15, 3.14, -0.17, 1.16]
[0.17, -2.15, -1.16, -3.14]
[0.17, -2.15, -1.16, -3.14, 11.1]
Elemento: 2.15
Elemento: 3.14
Elemento: -0.17
Elemento: 1.16
[-0.17, 1.16, 2.15, 3.14, 22.2]
Elemento: -0.17
Elemento: 1.16
Elemento: 2.15
Elemento: 3.14
Elemento: 22.2
```

ESEMPIO PLURI-LINGUAGGIO: Set in C#

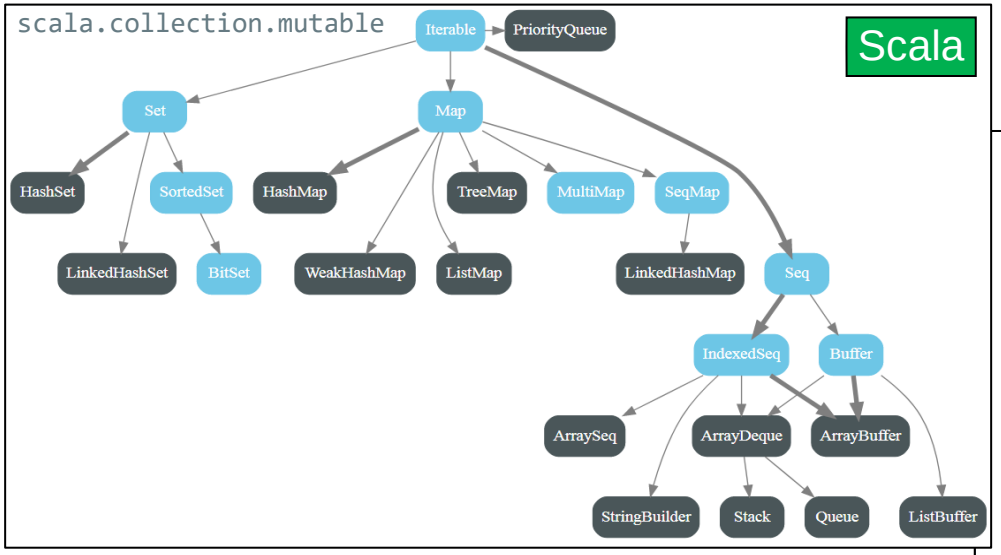
- L'interfaccia **`ISet`** definisce un *insieme*
- Non esistono factory internalizzate: le implementazioni *immodificabili* sono definite in un altro namespace
- Implementazioni *modificabili* di `ISet` sono **`HashSet`** e **`SortedSet`** (quest'ultima è un insieme *ordinato*)
- L'iteratore implicito del *foreach* naviga il set in modo arbitrario: segue però l'ordine nel caso di `SortedSet`
- Notare il costruttore per copia: tutte le collection (qui `SortedSet` in `s3`) accettano come argomento un'altra collection (qui, un `ISet`) di cui copiano (in modo shallow) tutti gli elementi
- NB: la `ToString` ereditata stampa una stringa standard inutile → meglio usare `String.Join` (che sfrutta l'iteratore)

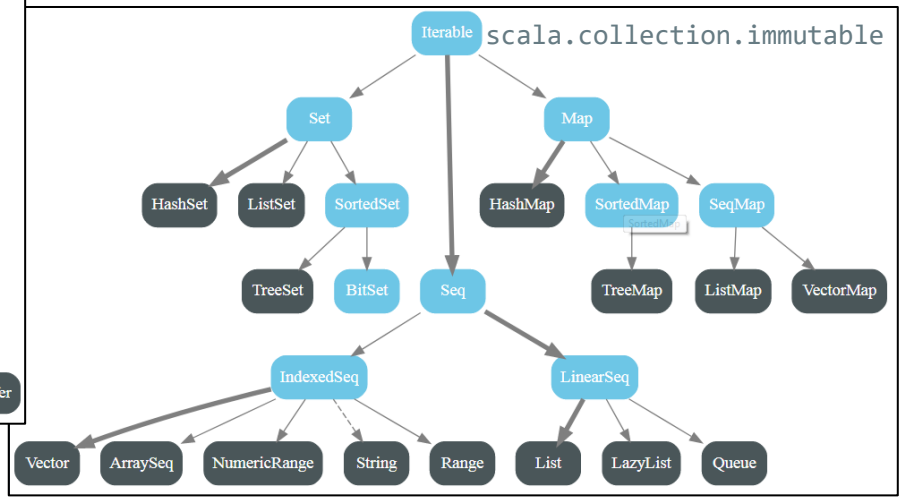
```
public static void Main()
{
    ISet<double> s1 = new HashSet<double>{3.14, 2.15, 1.16, -0.17};
    ISet<double> s2 = new HashSet<double>();
    s2.Add(-3.14); s2.Add(-2.15);
    s2.Add(-1.16); s2.Add(0.17);
    Console.WriteLine(String.Join(",", s1));
    Console.WriteLine(String.Join(",", s2));
    // s1.add(22.2); // UnsupportedOperationException
    s2.Add(11.1);
    Console.WriteLine(String.Join(",", s2));
    foreach (double d in s1) Console.WriteLine("Elemento: " + d);
    ISet<double> s3 = new SortedSet<double>(s1);
    s3.Add(22.2);
    Console.WriteLine(String.Join(",", s3));
    foreach (double d in s3) Console.WriteLine("Elemento: " + d);
```

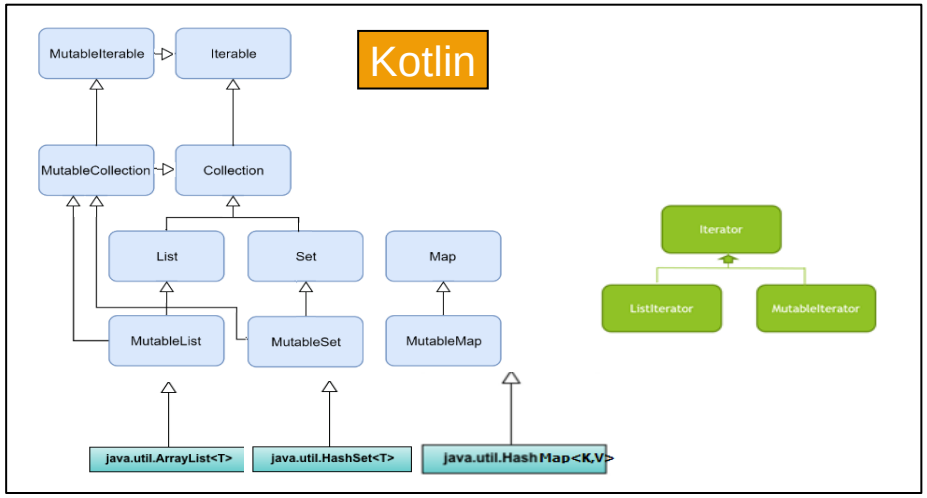
```
3.14,2.15,1.16,-0.17
-3.14,-2.15,-1.16,0.17
-3.14,-2.15,-1.16,0.17,11.1
Elemento: 3.14
Elemento: 2.15
Elemento: 1.16
Elemento: -0.17
-0.17,1.16,2.15,3.14,22.2
Elemento: -0.17
Elemento: 1.16
Elemento: 2.15
Elemento: 3.14
Elemento: 22.2
```

C#

COLLECTION FRAMEWORK: UN ESEMPIO PLURI-LINGUAGGIO







ESEMPIO PLURI-LINGUAGGIO: Set in Scala

Scala

```
import scala.collection.mutable._;

object EsempioSet1{

  def main(args: Array[String]) : Unit = {
        val s1 = scala.collection.immutable.Set(3.14, 2.15, 1.16, -0.17);
        val s2 : Set[Double] = HashSet[Double]();
        s2.add(-3.14); s2.add(-2.15);
        s2.add(-1.16); s2.add(0.17);
        println(s1);
        println(s2);
        // s1.add(22.2); // UnsupportedOperationException
        s2.add(11.1);
        println(s2);
        for (d <- s1) println("Elemento: " + d);
        //
        // Sotto: toSeq converte in sequenza, che l'operatore :_* trasforma in varargs
        val s3 : Set[Double] = LinkedHashSet(s1.toSeq :_*); // la factory si aspetta dei vargargs, non una collection
        s3.add(22.2);
        println(s3);
        for (d <- s3) println("Elemento: " + d);
```

- Il tratto **`Set`** definisce un *insieme* (modificabile o immutabile, dipende dal package importato: di default sarebbe immutabile, quindi qui è importato l'altro package)
- La factory internalizzata **`Set(…)`** costruisce e inizializza il set
- Implementazioni di `Set` sono **`HashSet`** e **`LinkedHashSet`** (quest'ultima preserva l'ordine *di inserimento*: non è una ordinata in senso totale di per sé, ma è sfruttabile a tale scopo)
- L'iteratore implicito del *foreach* naviga il set in modo arbitrario: garantisce invece di seguire l'ordine nel caso di `LinkedHasSet`
- Il costruttore per copia qui vuole l'elenco degli argomenti (varargs), ottenibile da una sequenza (da cui il metodo `toSeq`) tramite lo speciale operatore Scala **`:_*`**

ESEMPIO PLURI-LINGUAGGIO: Set in Kotlin

Kotlin

```
fun main(args: Array<String>) : Unit {
        val s1 : Set<Double> = setOf(3.14, 2.15, 1.16, -0.1
        val s2 : MutableSet<Double> = HashSet<Double>();
        s2.add(-3.14); s2.add(-2.15);
        s2.add(-1.16); s2.add(0.17);
        println(s1);
        println(s2);
        // s1.add(22.2); // UnsupportedOperationException
        s2.add(11.1);
        println(s2);
        for (d in s1) println("Elemento: " + d);
        val s3 : MutableSet<Double> = LinkedHashSet<Double>(s1);
        s3.add(22.2);
        println(s3);
        for (d in s3) println("Elemento: " + d);
```

- L'interfaccia **`Set`** definisce un *insieme immutabile*, mentre l'interfaccia **`MutableSet`** definisce un *insieme modificabile*
- Metodo factory per `Set`: **`setOf`** (non esiste un metodo factory per set modificabili)
- Due utili implementazioni di `MutableSet` sono **`HashSet`** e **`LinkedHashSet`** (quest'ultima preserva l'ordine di *inserimento,* ma non è ordinata di per sé)
- L'iteratore implicito del *foreach* naviga il set in modo arbitrario: segue invece l'ordine nel caso di `LinkedHashSet`
- Anche qui, il costruttore per copia (nell'esempio, di `LinkedHasSet`) accetta come argomento un'altra collection di cui copia (in modo shallow) tutti gli elementi.

CINQUE PICCOLI ESERCIZI

a) Uso di **`Set`** per operare su un insieme di elementi
   - esempio: un elenco di parole senza doppioni (Esercizio n.1)

b) Uso di **`List`** per operare su una sequenza di elementi
   - scambiando due elementi nella sequenza (Esercizio n.2)
   - o iterando dal fondo con un iteratore di lista (Esercizio n.3)

c) Uso di **`Map`** per fare una tabella di elementi e contarli
   - esempio: contare le occorrenze di parole (Esercizio n.4)

d) Uso di **`SortedMap`** per creare un elenco ordinato
   - idem, ma creando poi un elenco ordinato (Esercizio n.5)

CINQUE PICCOLI ESERCIZI: QUALI IMPLEMENTAZIONI ?

- Istintivamente, verrebbe subito da chiedersi *quali implementazioni usare* nei vari esercizi…
- …ma sarebbe sbagliato: per usare le collezioni *non occorre* conoscere l'implementazione, *basta attenersi alle interfacce!*
  - anche per pianificare il collaudo, in effetti, non serve saperlo.. ☺
- Quindi, ora ragioneremo sui vari esercizi e li imposteremo *lasciando volutamente in bianco la fase di costruzione,* in modo da non legarci ad alcuna implementazione
  - *solo dopo* li riprenderemo in mano scegliendo fra diverse implementazioni possibili, valutando analogie e differenze

ESERCIZIO 1 – `Set`

- Il problema: analizzare un insieme di parole
  - ad esempio, gli argomenti della riga di comando
- e specificatamente:
  - stampare tutte le parole *duplicate*
  - stampare il *numero* di parole *distinte*
  - stampare *l'elenco* delle parole *distinte*

Visto che interessano le parole *distinte*

- ossia, senza duplicati

la struttura dati più adatta è certamente un **`Set`**

- garantisce di suo l'assenza di duplicati
- il suo metodo **`add`** li elimina a priori: tentando di inserire un elemento già esistente, restituisce **`false`**

ESERCIZIO 1 – Set

Java

```
import java.util.*;

public class FindDups {

 public static void main(String args[]){

   Set<String> s = new 

   for (int i=0; i<args.length; i++)
    if (!s.add(args[i]))
      System.out.println("Parola duplicata: " + args[i]);

   System.out.println(s.size() + " parole distinte: "+s);
 }
}
```

Sceglieremo dopo l'implementazione

```
>java FindDups Io sono Io esisto Io parlo
Parola duplicata: Io
Parola duplicata: Io
4 parole distinte: [Io, parlo, esisto, sono]
```

nessun ordine

ESERCIZIO 1 – `Set`

- Come si vede, la **`toString`** di default dei **`Set`** stampa gli elementi nella forma `[e1, e2, …]`
- Volendo personalizzare la stampa, si può usare un ciclo

  anziché

```
System.out.println(s.size() + " parole distinte: " +s);
```

  si può scrivere ad esempio

```
for (String st : s) System.out.print(st + " ");
```

```
>java FindDups Io sono Io esisto Io parlo
Parola duplicata: Io
Parola duplicata: Io
Io parlo esisto sono
```

ESERCIZIO 1 – Set

- Lo stesso esempio negli altri linguaggi:

C#

```
ISet<string> s = new  implementazione da stabilire
for (int i=0; i<args.Length; i++)
 if (!s.Add(args[i]))
    Console.WriteLine("Parola duplicata: " + args[i]);
Console.WriteLine(s.Count + " parole distinte: " + String.Join(",", s));
```

Scala

```
val s: Set[String] = new implementazione da stabilire
for (i <- 0 until arg.size)
 if (!s.contains(arg(i)))
    s += arg(i);
 else
    println("Parola duplicata: " + arg(i));
println(s.size.toString() + " parole distinte: "+s);
```

Kotlin

```
val s: MutableSet<String> = class to be selected
for (i in 0..args.size-1)
 if (!s.add(args[i]))
    println("Parola duplicata: " + args[i]);
println(s.size.toString() + " parole distinte: "+s);
```

ESERCIZIO 2 – `List`

- Il problema: scambiare due parole in una sequenza
  - ad esempio, due argomenti presi dalla riga di comando

"Scambiare" presuppone una nozione di *posizione*
- il **`Set`** non ha questa caratteristica

La struttura dati più adatta è una **`List`**
- come un array, ha una nozione di *sequenza di elementi* numerati a partire da zero
- serve una funzione accessoria **`swap`** per fare lo scambio

Java

```
static void swap(List<T> list, int i, int j) {
  T temp = list.get(i);
  list.set(i, list.get(j)); list.set(j, temp);
}
```

ESERCIZIO 2 – List

Java

```
public static void main(String args[]){
  List<String> list = new …
  for (int i=0; i<args.length; i++) list.add(args[i]);
  System.out.println(list);
  swap(list, 2, 3);
  System.out.println(list);
}
```

Sceglieremo dopo l'implementazione

```
java EsList cane gatto pappagallo
  canarino cane canarino pescerosso
[cane, gatto, pappagallo, canarino,
  cane, canarino, pescerosso]
[cane, gatto, canarino, pappagallo,
  cane, canarino, pescerosso]
```

Elementi n. 2 e 3 scambiati

ESERCIZIO 2 – List

- Lo stesso esempio negli altri linguaggi:

C#

```
        IList<string> list = new class to be chosen later
        for (int i=0; i<args.Length; i++) list.Add(args[i]);
        Console.WriteLine(String.Join(",", list));
        swap(list, 2, 3);
        Console.WriteLine(String.Join(",", list));

    }

    // ------------------------------

    static void swap<T>(IList<T> list, int i, int j)
        T temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }
```

Scala

```
    var list : Buffer[String] = new class to be selected

    for (i <- 0 until arg.length) list += arg(i);
    println(list);
    swap(list, 2, 3);
    println(list);
}

// ------------------------------

def swap[T](list : Buffer[T], i:Int, j:Int) : Unit = {
        val temp = list(i);
        list(i) = list(j);
        list(j) = temp;
}
```

Kotlin

```
        var list : MutableList<String> = suitable class to b

        for (i in 0..arg.size-1) list.add(arg[i]);
        println(list);
        swap(list, 2, 3);
        println(list);
}

fun <T> swap(list : MutableList<T>, i:Int, j:Int) : Unit {
    val temp = list.get(i);
    list.set(i, list.get(j));
    list.set(j, temp);
}
```

Da `Iterator` A `ListIterator`

- In alcuni linguaggi si introduce *l'iteratore di lista,* ***`ListIterator`***, che specializza **`Iterator`** per:
  - andare anche all'indietro (è una sequenza!) — **`previous`**
  - sapere se ci sono elementi precedenti — **`hasPrevious`**
  - avere un concetto di *indice* (come negli array)
  - tornare all' *indice precedente* — **`previousIndex`**
  - avanzare all'*indice successivo* — **`nextIndex`**
  - aggiungere un elemento alla posizione i-esima — **`add(int)`**
  - modificare l'elemento alla posizione i-esima — **`set(int)`**

Java Kotlin

- Altri linguaggi adottano soluzioni differenti, senza introdurre una nuova interfaccia ad hoc.

C# Scala

ESERCIZIO 3
List & ListIterator

- Schema tipico di iterazione a ritroso:

```
for(  ListIterator i = l.listIterator(l.size()) ;
      i.hasPrevious() ; ) {
   ...
}
```

Java

Per usare `hasPrevious`, occorre ovviamente iniziare dalla fine

- Esempio: riscrittura a rovescio degli argomenti

```
...
for( ListIterator i = l.listIterator(l.size()) ;
     i.hasPrevious() ; ) {
  System.out.print( i.previous() + " ");
}
```

Java

```
java EsListIt cane gatto cane canarino
canarino cane gatto cane
```

ESERCIZIO 3
`List` & `ListIterator`

- Lo stesso esempio negli altri linguaggi:
  - in C# non esiste un iteratore inverso: occorre rovesciare la lista → necessario farne una copia, per preservare l'originale
  - per farlo occorre però lavorare sulla classe **`List`**, in quanto l'interfaccia **`IList`** non dichiara a priori il metodo **`Reverse`** ☹

C#

```
IEnumerator<string> iter = list.GetEnumerator();
while (iter.MoveNext()) Console.WriteLine(iter.Current);
Console.WriteLine("---");

List<string> l = new List<string>(list);

l.Reverse();    // è una procedura, non restituisce nulla, lavora "in place"

IEnumerator<string> revIter = l.GetEnumerator();
while (revIter.MoveNext()) Console.WriteLine(revIter.Current);
Console.WriteLine("---");
// OPPURE
IEnumerable<string> revl = (list as IEnumerable<string>).Reverse();
IEnumerator<string> revIter2 = revl.GetEnumerator();
while (revIter2.MoveNext()) Console.WriteLine(revIter2.Current);
Console.WriteLine("---");
```

ESERCIZIO 3
`List` & `ListIterator`

- Lo stesso esempio negli altri linguaggi:
  - in Scala non esiste un'interfaccia ad hoc per l'iteratore inverso: si usa un iteratore standard «invertito», prodotto dall'apposito metodo **`reverseIterator`**
  - in Kotlin la situazione è identica a Java

Scala

```
val iter : Iterator[String] = list.iterator;
while (iter.hasNext) println(iter.next)
println("---")

val revIter : Iterator[String] = list.reverseIterator;
while (revIter.hasNext) println(revIter.next)
println("---")
```

Kotlin

```
val iter : Iterator<String> = list.iterator();
while (iter.hasNext()) println(iter.next());
println("---")

val revIter : ListIterator<String> = list.listIterator(list.size);
while (revIter.hasPrevious()) println(revIter.previous())
println("---")
```

ESERCIZIO 4 – `Map`

- Il problema: contare le occorrenze delle parole
  - ad esempio, degli argomenti della riga di comando

Si potrebbe usare una lista, ma sarebbe *inefficiente*

- dopo aver creato la lista con tutte le parole, bisognerebbe *scorrerla tutta di nuovo* per contare le occorrenze
- peggio ancora, servirebbero *tanti contatori diversi* (uno per ogni parola) e non si può sapere a priori quanti

Con una **`Map`** invece la cosa è semplice e lineare

- basta fare una tabella (parole, occorrenze)
- ogni riga una parola: a fianco, il numero di occorrenze
- quando si incontra una nuova parola, si inserisce una nuova riga e si scrive "numero di occorrenze = 1"

ESERCIZIO 4 – `Map`

- Serve una **`Map<String, Integer>`**

Java

  - ricorda: non si possono usare direttamente i *tipi primitivi* come **`int`**, occorre usare il corrispondente wrapper
  - però, si possono inserire/estrarre direttamente *valori primitivi* dalla tabella, grazie al boxing/unboxing automatico

parola	*occorrenze*

parola	*occorrenze*
cane	1

parola	*occorrenze*
cane	1
gatto	1

parola	*occorrenze*
cane	**2**
gatto	1

parola	*occorrenze*
cane	2
gatto	1
pesce	1

ESERCIZIO 4 – `Map`

Java

- Serve una **`Map<String, Integer>`**
  - ricorda: non si possono usare direttamente i *tipi primitivi* come **`int`**, occorre usare il corrispondente wrapper
  - però, si possono inserire/estrarre direttamente *valori primitivi* dalla tabella, grazie al boxing/unboxing automatico
- Il metodo **`put(`*`chiave,valore`*`)`** inserisce una nuova riga
  - NB: se c'è già una riga con la stessa chiave, la *sostituisce*
- Il metodo **`get(`*`chiave`*`)`** cerca una riga con quella chiave e se la trova restituisce il ***`valore`*** associato
  - se non la trova, restituisce **`null`**

ESERCIZIO 4 – Map

Java

```
import java.util.*;
public class ContaFrequenza {
 public static void main(String args[]) {
   Map<String,Integer> m = new _________;
   for (int i=0; i<args.length; i++) {
    Integer freq = m.get(args[i]);
    m.put(args[i], (freq==null ? 1 :freq + 1) );
   }
   System.out.println(m.size() + " parole distinte:");
   System.out.println(m);
 }
}
```

La **toString** stampa un elenco di coppie nella forma *key=value*

```
>java ContaFrequenza cane gatto cane pesce gatto gatto cane
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

ESERCIZIO 4 – Map
Variante: iterare sulla mappa ottenuta

- E se volessimo iterare per stampare noi gli elementi?
  - facile: basta recuperare l'insieme delle chiavi con **`keySet()`** e iterare su quello
  - si recupera poi con **`get`** l'elemento corrispondente a ogni chiave

```java
public static void myPrint(Map<String,Integer> m){
  Set<String> keys = m.keySet();
  for (String key: keys) {
       System.out.println(key + "\t" + m.get(key));
  }
}
```

Java

```
pesce    1
cane     3
gatto    3
```

ESERCIZIO 4 – Map
Variante: iterare sulla mappa ottenuta

- Alternativa: agire per righe
  - invece che recuperare l'insieme delle chiavi e iterare su quello, si può recuperare *l'insieme delle righe* con **`entrySet()`**
  - si evita così di usare **`get`** per recuperare gli elementi, seppur al (piccolo) prezzo di dover usare la *classe interna* **`Map.Entry`**

```
public static void myPrint2(Map<String,Integer> m){
   Set<Map.Entry<String,Integer>> rows = m.entrySet();
   for (Map.Entry row: rows) {
        System.out.println(row);
   }
}
```

Java

È la **`toString`** di **`Map.Entry`**

```
pesce=1
cane=3
gatto=3
```

ESERCIZIO 4 – Map

- Lo stesso esempio negli altri linguaggi:
  - in C# tutto è molto simile a Java, salvo che non esiste il metodo **`entrySet`**: possiamo però facilmente simularlo

C#

```
    IDictionary<string,int> m = new to be chosen later
    for (int i=0; i<args.Length; i++) {
     int freq = m.ContainsKey(args[i]) ? m[args[i]] : 0;
     m[args[i]] = freq+1;
    }
    Console.WriteLine(m.Count.ToString() + " parole distinte:");
    Console.WriteLine(String.Join(",", m));           // stampa esattamente come sotto
    MyPrint(m);
    Console.WriteLine(String.Join(",", entrySet(m))); // stampa esattamente come sopra
}

public static void MyPrint(IDictionary<string,int> m){
    ICollection<string> keys = m.Keys;
    foreach (string key in keys) {
        Console.WriteLine(key + "\t" + m[key]);
    }
}

public static ISet<KeyValuePair<string,int>> entrySet(IDictionary<string,int> m) {
// simula la entrySet di Java
    ISet<KeyValuePair<string,int>> result = new HashSet<KeyValuePair<string,int>>();
    foreach(KeyValuePair<string,int> entry in m) result.Add(entry);
    return result;
}
```

ESERCIZIO 4 – Map

- Lo stesso esempio negli altri linguaggi:
  - in Scala di base le cose sono assai simili a Java…

Scala

```
    val m : Map[String,Int] = new to be chosen later

    for (i <- 0 until arg.length) {
         val freq : Option[Int] = m.get(arg(i));
         m += arg(i)->(freq.getOrElse(0)+1);
    }
    println(m.size.toString() + " parole distinte:");
    println(m);
    //
    myPrint(m);
    myPrintEntries(m);
    //
    println(entrySet(m))
}

def myPrint(m : Map[String,Int]) : Unit = { // stampa "chiave \t valore"
    val keys: scala.collection.Set[String] = m.keySet;
    for (key <- keys) {
        println(key + "\t" + m.get(key).get);
    }
}
```

ESERCIZIO 4 – Map

- Lo stesso esempio negli altri linguaggi:
  - .. tranne per il fatto che non esiste la classe **`Entry`**: le righe sono tuple della forma (key, value) su cui *si può iterare direttamente*
  - analogamente non esiste il metodo **`entrySet`**, ma possiamo simularlo con una funzione custom (che però ha bisogno di un set ausiliario su cui appoggiare l'implementazione)

```scala
def myPrintEntries(m : Map[String,Int]) : Unit = { // stampa "(chiave, valore)"
   val iter : Iterator[(String,Int)] = m.iterator;
   // itera sulle tuple (entries), ma non esiste Entry come classe
   while (iter.hasNext) println(iter.next);
}

def myPrintValues(m : Map[String,Int]) : Unit = { // stampa solo il valore
   val values: Iterable[Int] = m.values;
   for (value <- values) println(value);
}
```

Scala

```scala
def entrySet(m : Map[String,Int]) : Set[(String,Int)] = {
   val result = new HashSet[(String,Int)]();
   m.foreach(it => result += it)
   return result;
}
```

Scala

ESERCIZIO 4 – Map

- Lo stesso esempio negli altri linguaggi:
  - in Kotlin, come in Scala, le cose sono assai simili a Java

```
val m : MutableMap<String,Int> = to be chosen later

for (i in 0..arg.size-1) {
     val freq = m.get(arg[i]);
     m.put( arg[i], (if(freq==null) 1 else freq+1));
}
println(m.size.toString() + " parole distinte:");
println(m);
//
myPrint(m);
myPrint2(m);
```

Kotlin

```
fun myPrintEntries(m : Map<String,Int>) : Unit { // stampa "(chiave, valore)"
    val iter : Iterator<Map.Entry<String,Int>> = m.iterator(); // itera sulle tuple (entries)
    while (iter.hasNext()) println(iter.next());
}

fun myPrintValues(m : Map<String,Int>) : Unit { // stampa solo il valore
    val values: Iterable<Int> = m.values;
    for (value in values) println(value);
}

fun myPrint2(m : Map<String,Int>) : Unit {
    val rows : Set<Map.Entry<String,Int>> = m.entries;
    for (row in rows) println(row);
}
```

Kotlin

ESERCIZIO 4 – `Map`

- Lo stesso esempio negli altri linguaggi:
  - in Kotlin, come in Scala, le cose sono assai simili a Java
  - pur esistendo la classe **`Entry`**, come in Java, le righe si possono anche esprimere come tuple della forma (key, value) su cui *si può iterare direttamente,* come in Scala

Kotlin

```
fun myPrint(m : Map<String,Int>) : Unit {
    for ((key,value) in m) {
        println(key + "\t" + value);
    }
}

fun myPrintAlternative(m : Map<String,Int>) : Unit {
    // identica a myPrint, solo più tradizionale come codice
    val keys = m.keys;
    for (key in keys) {
        println(key + "\t" + m.get(key));
    }
}
```

ESERCIZIO 5 – SortedMap

- Variante con mappa ordinata

Java

```
import java.util.*;
public class ContaFrequenza {
 public static void main(String args[]) {
    SortedMap<String,Integer> m = new _________;
    for (int i=0; i<args.length; i++) {
     Integer freq = m.get(args[i]);
     m.put(args[i], (freq==null ? 1 :freq + 1) );
    }
    System.out.println(m.size() + "
    System.out.println(m);
 }
}
```

Nelle versioni *Sorted*, l'iteratore segue l'ordine *delle chiavi*: toString, di conseguenza, anche

```
>java ContaFrequenza cane gatto cane pesce gatto gatto cane
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

```
>java ContaFrequenzaOrd cane gatto cane pesce gatto gatto cane
3 parole distinte: {cane=3, gatto=3, pesce=1}
```

ESERCIZIO 5 – SortedMap

- Lo stesso esempio negli altri linguaggi:
  - in C# non esiste un'interfaccia **ISortedDictionary**: quindi, si mantiene l'interfaccia **IDictionary**, scegliendo poi implementazioni *Sorted* (**SortedDictionary** o **SortedList**, v.oltre)
  - in Scala tutto è molto simile a Java (si usa **SortedMap**)

Scala

```
    val m : SortedMap[String,Int] = to be chosen later

    for (i <- 0 until arg.length) {
         val freq : Option[Int] = m.get(arg(i));
         m += arg(i)->(freq.getOrElse(0)+1);
    }
    println(m.size.toString() + " parole distinte:");
    println(m);
    //
    myPrint(m);
    myPrintEntries(m);
    //
    println(entrySet(m))
}

def myPrint(m : SortedMap[String,Int]) : Unit = { // stampa "chiave \t valore"
    val keys: scala.collection.Set[String] = m.keySet;
    for (key <- keys) {
        println(key + "\t" + m.get(key).get);
    }
}
```

ESERCIZIO 5 – `SortedMap`

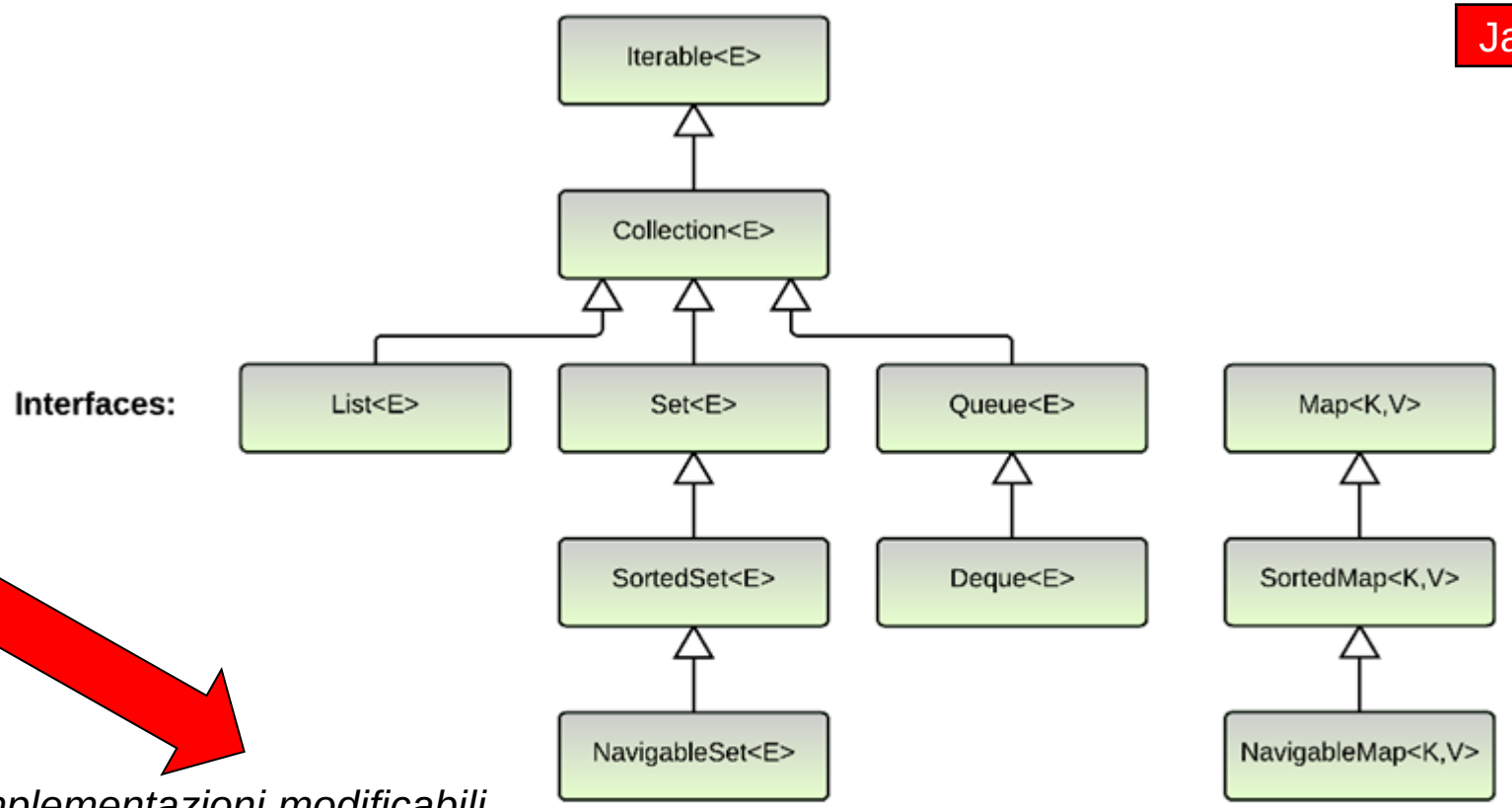
- Lo stesso esempio negli altri linguaggi:
  - in Kotlin non esiste un'interfaccia **`SortedMap`**
  - si usa un'implementazione che preservi l'ordine di inserimento (es. **`LinkedHashMap`**) per costruire una nuova mappa *già ordinata*
  - per farlo, si può sfruttare il metodo **`sorted()`** che restituisce una *copia ordinata* delle collezione su cui viene invocato (in questo caso, il set delle chiavi)

Kotlin

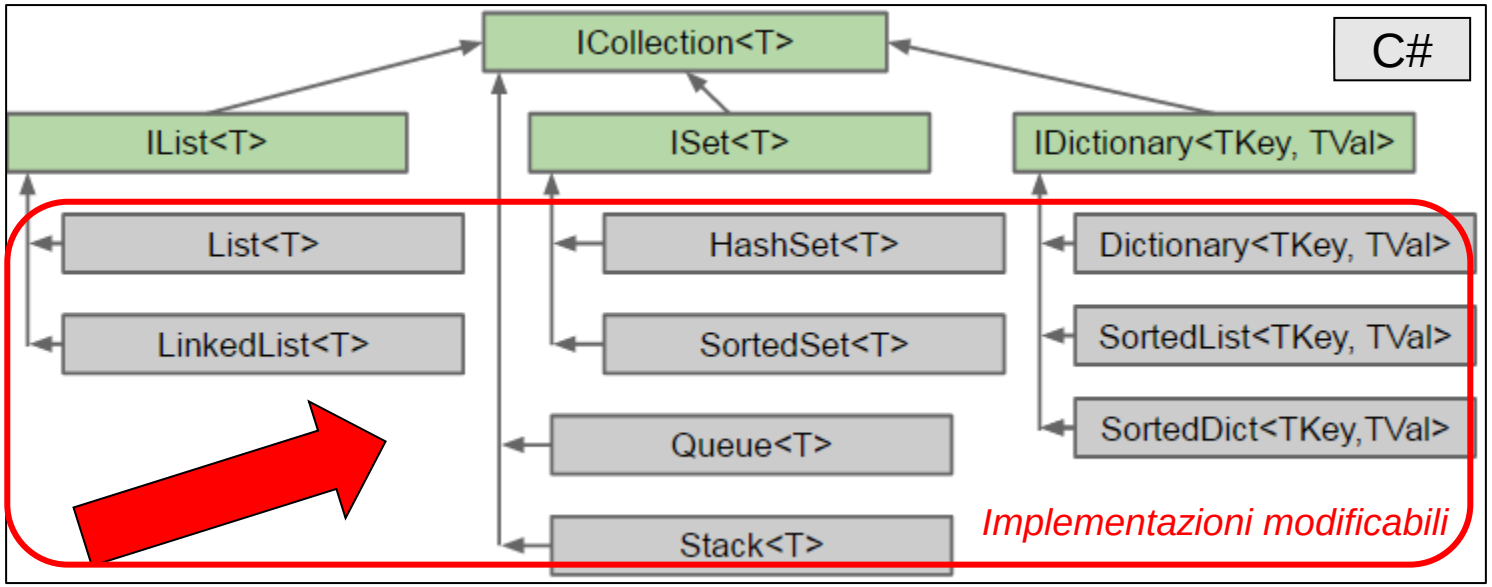
```
val ms : MutableMap<String,Int> = LinkedHashMap<String,Int>();
for(key in m.keys.sorted()) ms[key] = m[key]!!
//
println(m.size.toString() + " parole distinte:");
println(m);
//
println(ms.size.toString() + " parole distinte e ordinate:");
println(ms);
```

DALLE INTERFACCE ALLE IMPLEMENTAZIONI

Java



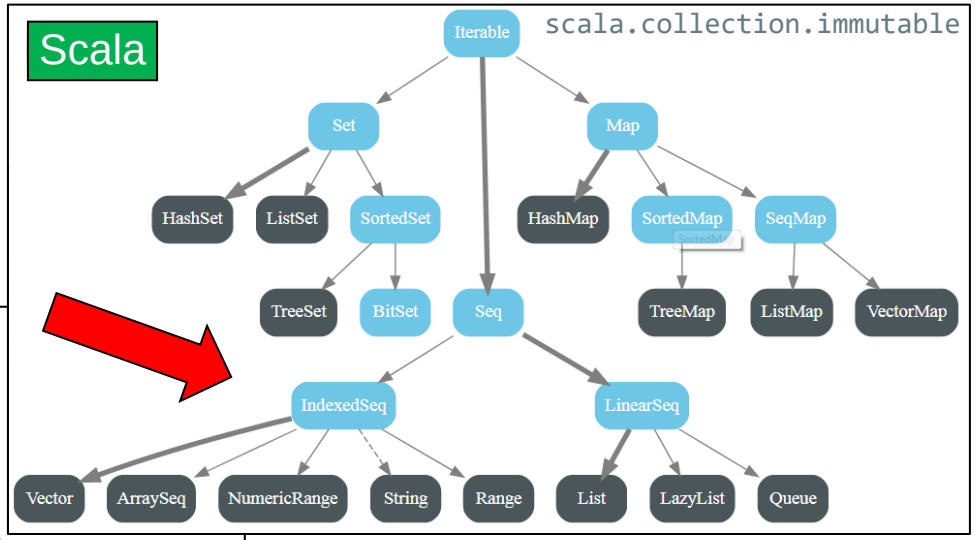
DALLE INTERFACCE ALLE IMPLEMENTAZIONI: C#

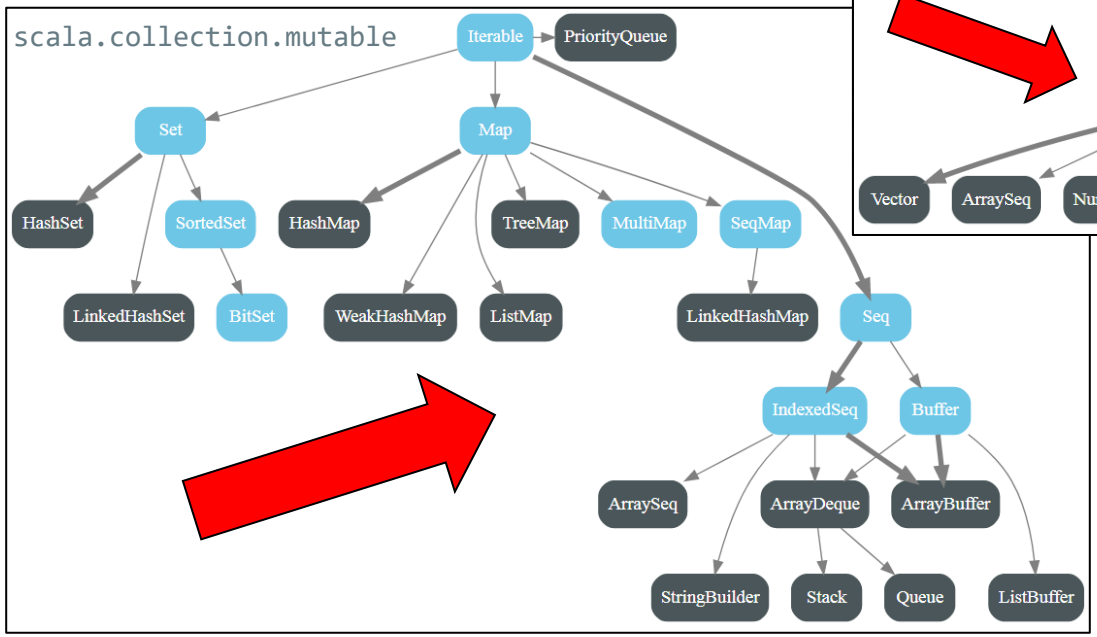


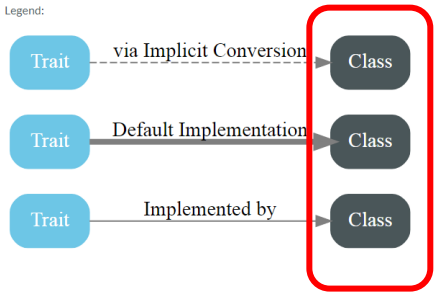
Le implementazioni *immodificabili* si trovano in **`System.Collections.Immutable`**, che definisce classi come **`ImmutableList`**, **`ImmutableSortedDictionary`**, etc.

DALLE INTERFACCE ALLE IMPLEMENTAZIONI: SCALA

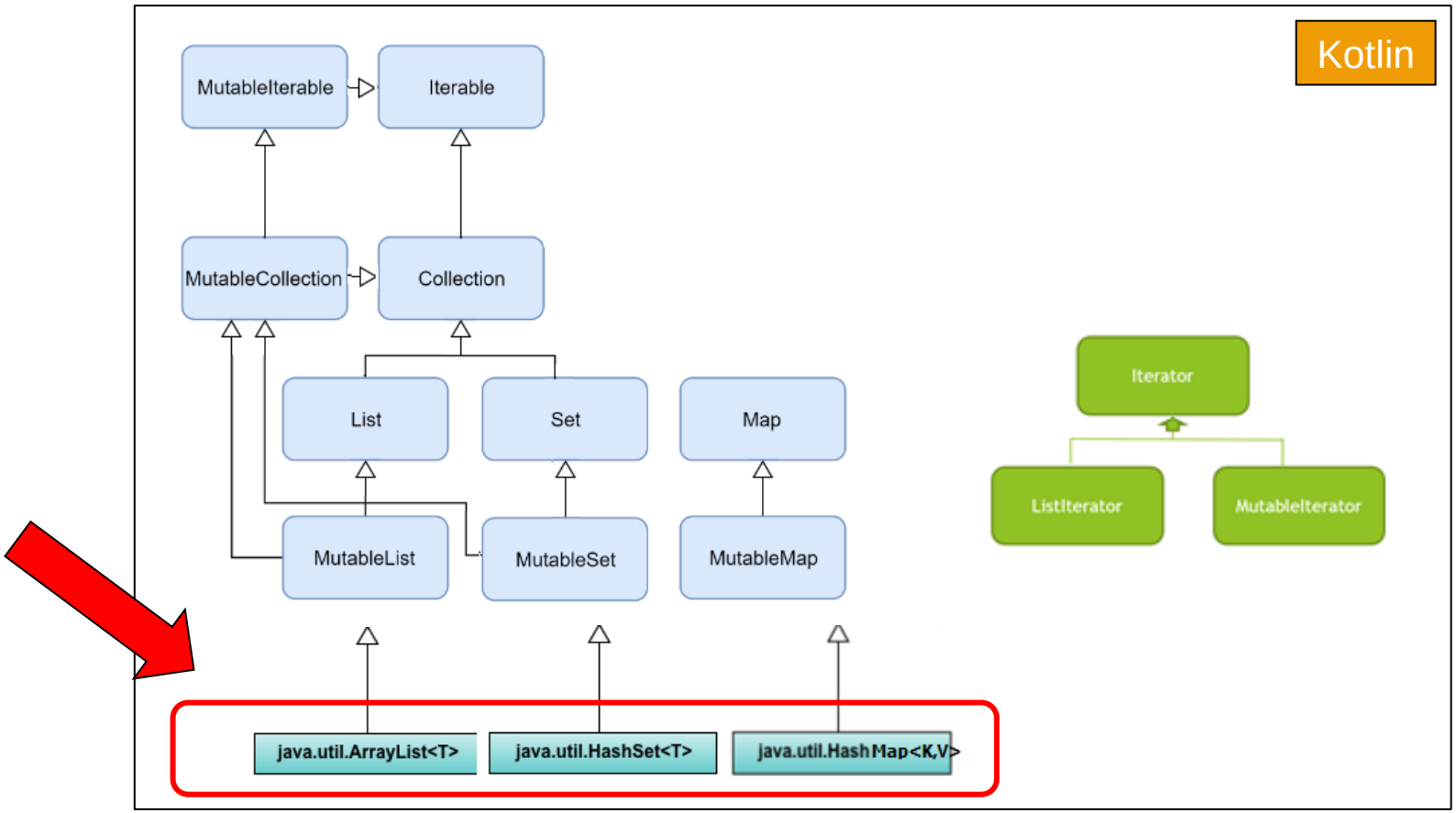
NB: questi diagrammi non seguono lo standard UML







DALLE INTERFACCE ALLE IMPLEMENTAZIONI: KOTLIN



COLLECTION FRAMEWORK: CROSS-LANGUAGE OVERVIEW

- Tabella comparativa
  - in Scala, di default, è importata la versione *immutable*

		Set		List		Map	
		Mutable	*Immutable*	*Mutable*	*Immutable*	*Mutable*	*Immutable*
Java	*interfaces*	`Set`	`Set`	`List`	`List`	`Map`	`Map`
	classes	`HashSet` `TreeSet`	(Set.of)	`ArrayList` `LinkedList`	(List.of)	`HashMap` `TreeMap`	(Map.of)
C#	*interfaces*	`ISet`	`IImmutableSet`	`IList`	`IImmutableList`	`IDictionary`	`IImmutableDictionary`
	classes	`Set` `HashSet` `SortedSet`	`ImmutableSet` `ImmutableHashSet`	`List` `SortedList`	`ImmutableList`	`Dictionary` `SortedDictionary` `SortedList`	`ImmutableDictionary` `ImmutableSortedDictionary`
Scala	*traits*	`Set`	`Set`	`Seq, Buffer`	`Seq, LinearSeq`	`Map`	`Map`
	classes	`HashSet` `LinkedHashSet`	`HashSet` `ListSet` `TreeSet`	`ArrayBuffer` `ListBuffer`	`List` `LazyList` `Queue`	`HashMap` `TreeMap,` `ListMap, …`	`HashMap` `TreeMap,` `ListMap, …`
Kotlin	*interfaces*	`MutableSet`	`Set`	`MutableList`	`List`	`MutableMap`	`Map`
	classes	`HashSet`	`HashSet`	`ArrayList`	`ArrayList`	`HashMap` `TreeMap` `ListMap, …`	`HashMap` `TreeMap` `ListMap, …`

DALLE INTERFACCE ALLE IMPLEMENTAZIONI: JAVA

Java

General purpose Implementations

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

- Implementazioni fondamentali (modificabili):
  - per **`Set`**: **`HashSet`**, **`TreeSet`**, **`LinkedHashSet`**, **`EnumSet`**
  - per **`List`**: **`ArrayList`**, **`LinkedList`**
  - per **`Map`**: **`HashMap`**, **`TreeMap`**, **`LinkedHashMap`**, **`EnumMap`**
  - per **`Queue`**: **`ArrayBlockingQueue`**, **`PriorityQueue`**, **`LinkedList`**
  - per **`Deque`**: **`ArrayDeque`**, **`LinkedList`**
- Le implementazioni *immodificabili* non corrispondono a classi pubbliche: sono prodotte solo dai factory methods **`List.of`**, **`Set.of`**, etc.

COLLEZIONI: LINEE GUIDA

Quali usare?

- Regole generali per **`Set`** e **`Map`**
  - se serve *l'ordinamento*, necessariamente **`TreeMap`** e **`TreeSet`** che implementano le sotto-interfacce **`SortedMap`** e **`SortedSet`**
  - altrimenti, **`HashMap`** e **`HashSet`** sono *nettamente più efficienti* (tempo di esecuzione costante anziché log(N))
- Regole generali per **`List`**
  - di norma meglio **`ArrayList`**, che ha *tempo di accesso costante* (anziché lineare con la posizione) perché è realizzata su array
  - preferire invece **`LinkedList`** se l'operazione più frequente è l'aggiunta in testa o l'eliminazione di elementi in mezzo alla lista.

COLLEZIONI: LINEE GUIDA

Quali usare? *(continua)*

- Implementazioni specifiche per **`Set`** e **`Map`**:
  - se serve un *ordine precidibile di iterazione*, **`LinkedHashMap`** e **`LinkedHashSet`**

    This implementation maintains a doubly-linked list through all of its entries. The *iteration ordering* is the *insertion-order*, even if an element is re-inserted.

  - caso particolare: se gli elementi del set o le chiavi della mappa sono *enumerativi*, utile scegliere **`EnumMap`** ed **`EnumSet`**

    The internal implementation is *extremely compact and efficient.* Iteration reflects *the natural ordering* of enum constants.

COLLEZIONI: COSTRUZIONE

- Le collezioni concrete si costruiscono
  - *vuote*, con un costruttore
  - *già inizializzate*, con appositi inizializzatori o metodi factory statici (in alcuni linguaggi, ciò si applica solo a collezioni *immodificabili*)
- Premesso che ogni linguaggio al riguardo fa le sue scelte, tipicamente:
  - i costruttori producono una collection inizialmente vuota
  - ulteriori *costruttori per copia* producono una collection a partire da un'altra (anche di diverso tipo) fornita come argomento
  - collezioni pre-inizializzate sono tipicamente ottenute tramite *metodi factory statici* (in alcuni linguaggi, solo per collezioni *immodificabili*)

COLLEZIONI: COSTRUZIONE

- In Java

Java

  - tutte le classi-collection definiscono un costruttore a zero argomenti che produce una collezione vuota (modificabile)
  - tutte le classi-collection definiscono un costruttore per copia che accetta come argomento *un'altra* **`Collection`**
  - apposite factory internalizzate producono collezioni pre-popolate *immodificabili* per i casi standard: **`List.of`**, **`Set.of`**, **`Map.of`**

- In Kotlin

Kotlin

  - costruzione base come sopra
  - collezioni pre-popolate sono prodotte da factory method, sia per le versioni *modificabili* - **`listOf(…)`**, **`setOf(…)`**, **`mapOf(…)`** - che *immodificabili* - **`mutableListOf(…)`**, **`mutableSetOf(…)`**, etc.
  - per uniformità, tale approccio vale anche per gli array: **`arrayOf(…)`**

COLLEZIONI: COSTRUZIONE

- In Scala

Scala

  - costruzione base come sopra
  - collezioni pre-popolate sono prodotte da factory method appositi che hanno lo stesso nome della classe - **`List(…)`**, **`Set(…)`**, **`Map(…)`** - che esistono sia in versione *modificabile* che *immodificabile* (a seconda del package importato)
  - per uniformità, tale approccio vale anche per gli array: **`Array(…)`**

- In C#

C#

  - costruzione base come sopra
  - collezioni pre-popolate (modificabili) sono prodotte dal costruttore + inizializzatori della forma array-like **`{…}`**, che vale naturalmente anche per gli array
  - collezioni *immodificabili* disponibili in apposito namespace ad hoc

COSTRUZIONE DI COLLEZIONI: ESEMPI IN JAVA

- Costruzione di liste (modificabili)

Java
```
List<String> l1 = new LinkedList<String>();
l1.add("Bologna"); l1.add("Modena"); l1.add("Parma");
System.out.println(l1);
```

Java
```
List<String> l2 = new ArrayList<String>();
l2.add("Ferrara"); l2.add("Ravenna"); l2.add("Forlì");
System.out.println(l2);
```

Java
```
var l3 = new ArrayList<String>();
l3.add("Piacenza"); l3.add("Cesena"); l3.add("Reggio");
System.out.println(l3);
```

Type inference

```
[Bologna, Modena, Parma]
[Ferrara, Ravenna, Forlì]
[Piacenza, Cesena, Reggio]
```

COSTRUZIONE DI COLLEZIONI: ESEMPI IN JAVA

- Costruzione di liste immodificabili

```
List<String> l1 = List.of("Bologna", "Modena", "Parma")
System.out.println(l1);
```

Java

Metodo factory

Type inference

```
var l2 = List.of("Ferrara", "Ravenna", "Forlì");
System.out.println(l2);
```

Metodo factory

```
[Bologna, Modena, Parma]
[Ferrara, Ravenna, Forlì]
```

IMPORTANTE: il tipo prodotto dalla factory `List.of` non è una «classica» lista: è un'implementazione ad hoc, *value-based*, che potrebbe anche cambiare nel tempo. Non fare ipotesi su di essa!

COSTRUZIONE DI COLLEZIONI: ESEMPI IN JAVA

- Costruzione di set (modificabili)

Java
```
Set<String> s1 = new HashSet<String>();
s1.add("Bologna"); s1.add("Modena"); s1.add("Parma");
System.out.println(s1);
```

Java
```
Set<String> s2 = new TreeSet<String>();
s2.add("Bologna"); s2.add("Modena"); s2.add("Parma");
System.out.println(s2);
```
Insieme ordinato

Java
```
var s3 = new HashSet<String>();
s3.add("Piacenza"); s3.add("Cesena"); s3.add("Reggio");
System.out.println(s3);
```
Type inference

```
[Parma, Bologna, Modena]
[Bologna, Modena, Parma]
[Reggio, Piacenza, Cesena]
```
Insieme ordinato

COSTRUZIONE DI COLLEZIONI: ESEMPI IN JAVA

- Costruzione di set immodificabili

```
Set<String> s1 = Set.of("Bologna", "Modena", "Parma");
System.out.println(s1);
```

Java

Metodo factory

```
var s2 = Set.of("Ferrara", "Ravenna", "Forlì");
System.out.println(s2);
```

Type inference

Metodo factory

```
[Bologna, Parma, Modena]
[Ravenna, Forlì, Ferrara]
```

IMPORTANTE: il tipo prodotto dalla factory `Set.of` non è un «classico» set: è un'implementazione ad hoc, *value-based*, che potrebbe anche cambiare nel tempo. Non fare ipotesi su di esso!

NB: non esiste una factory `SortedSet.of`
Si può facilmente ovviare mediante una costruzione per copia, secondo il pattern `new TreeSet<…>(Set.of(…));`

COSTRUZIONE DI COLLEZIONI: ESEMPI IN JAVA

- Costruzione di mappe (modificabili)

```
Map<String,Integer> m = new HashMap<String,Integer>();
m.put("Bologna", 395416);
m.put("Modena", 189013);
m.put("Parma", 200455);
System.out.println(m);
```

Java

```
Map<String,Integer> m2 = new TreeMap<String,Integer>();
...
```

Mappa ordinata

```
var m3 = new HashMap<String,Integer>();
...
```

Type inference

```
{Parma=200455, Bologna=395416, Modena=189013}
{Bologna=395416, Modena=189013, Parma=200455}
{Parma=200455, Bologna=395416, Modena=189013}
```

Ordinata per chiave

COSTRUZIONE DI COLLEZIONI: ESEMPI IN JAVA

- Costruzione di mappe immodificabili

Java

```
Map<String,Integer> m1 = Map.of(
   "Bologna", 395416,
   "Modena", 189013,
   "Parma", 200455);
System.out.println(m1);
```

Metodo factory

Le entry si esprimono a coppie (chiave, valore) alternati

Type inference

```
var m2 = Map.of(
   "Bologna", 395416,
   "Modena", 189013,
   "Parma", 200455);
System.out.println(m2);
```

Anche qui, la classe restituita non è la classica implementazione di Map, è una classe ad hoc: non fare ipotesi su di essa!

Anche qui non esiste una factory `SortedMap.of`
Si può ovviare in modo analogo ai set
`new TreeMap<…>(Map.of(…));`

```
{Bologna=395416, Parma=200455, Modena=189013}
{Bologna=395416, Parma=200455, Modena=189013}
```

SCORCIATOIE LINGUISTICHE

- In generale, la costruzione di strutture tipizzate richiede di ripetere due volte il tag di tipo:

```
List<String> l1 = new ArrayList<String>();
Map<String,Integer> m1 = new HashMap<String,Integer>();
```

Java

- Poiché ciò è verboso, i linguaggi introducono *scorciatoie*
  - l'uso della type inference e della keyword **`val`**/**`var`** — Scala, Kotlin
  - l'uso della type inference e della keyword **`var`** — Java, C#

In queste notazioni il tipo rimane specificato *solo nella new (a destra),* venendo invece *omesso nel riferimento* (a sinistra), per type inference

```
var l1 = new ArrayList<String>();
var m1 = new HashMap<String,Integer>();
```

Java, ~C#, ~Scala, ~Kotlin

JAVA: DIAMOND OPERATOR

- Java introduce anche la possibilità inversa, quella di *omettere il tipo nella new* se è già specificato nel riferimento
- È il cosiddetto *diamond operator* **<>**

```
List<String> l1 = new ArrayList<>();
Map<String,Integer> m1 = new HashMap<>();
```

Java

- NB: non equivale ad abolire il tipo, né a scriverci **`Object`** !
  - equivale semplicemente a immaginare scritto lì *lo stesso tipo già indicato nel riferimento a sinistra*
- Chiaramente, è alternativo all'uso di **`val`**/**`var`**
  - per forza: da qualche parte, il tipo deve pur esserci.. ☺

COLLEZIONI IMMODIFICABILI: ASPETTI NOTEVOLI

- In Java, le collezioni *immodificabili* hanno i seguenti vincoli:
  - i metodi di modifica (**`add`**, **`put`**, **`remove`**) lanciano eccezione
  - non sono consentiti elementi o chiavi **`null`**
  - l'ordine di esplorazione degli iteratori è *indefinito* e può *cambiare nel tempo*: *non* fare ipotesi su di esso!
  - le classi concrete prodotte da questi metodi ***non*** *sono le classiche HashSet, ArrayList, etc,* ma *implementazioni ad hoc* che possono *cambiare nel tempo* → anche qui, NON fare ipotesi su di esse!
- In particolare, sono *value-based*
  - la factory si tiene il diritto di scegliere se *creare nuove istanze* o *restituire quelle già esistenti*, volta per volta
  - ergo, il test **`list1 == list2`** ha *risultato impredicibile:* non è dato sapere se due collezioni si riferiscano allo stesso oggetto!

COSTRUZIONE DI COLLEZIONI: ESEMPI IN C#

- Costruzione di liste (modificabili)

C#

```
ICollection<string> l1 = new LinkedList<string>();
l1.Add("Bologna"); l1.Add("Modena"); l1.Add("Parma");
Console.WriteLine(String.Join("," , l1));
```

```
IList<string> l2 = new List<string>(); // ~array list
l2.Add("Ferrara"); l2.Add("Ravenna"); l2.Add("Forlì");
Console.WriteLine(String.Join("," , l2));
```

```
var l3 = new List<string>();
l3.Add("Piacenza"); l3.Add("Cesena"); l3.Add("Reggio");
Console.WriteLine(String.Join("," , l3));
```

Type inference

```
Bologna,Modena,Parma
Ferrara,Ravenna,Forlì
Piacenza,Cesena,Reggio
```

COSTRUZIONE DI COLLEZIONI: ESEMPI IN C#

- Costruzione di liste immodificabili
  - la lista si crea solo inizialmente vuota, con la costante **`Empty`**
  - successive «modifiche» producono via via nuove liste, senza mai modificare l'originale

C#

```
var list1 = ImmutableList<int>.Empty;
var list2 = list1.Add(14).Add(-15).Add(16);
Console.WriteLine("List1: " + String.Join(", ", list1));
Console.WriteLine("List2: " + String.Join(", ", list2));
var list3 = list2.Replace(-15,15);
Console.WriteLine("List3: " + String.Join(", ", list3));
Console.WriteLine("List2: " + String.Join(", ", list2));
```

```
List1:
List2: 14, -15, 16
List3: 14, 15, 16
List2: 14, -15, 16
```

COSTRUZIONE DI COLLEZIONI: ESEMPI IN C#

- Costruzione di set (modificabili)

C#

```
ISet<string> s1 = new HashSet<string>();
s1.Add("Bologna"); s1.Add("Modena"); s1.Add("Parma");
Console.WriteLine(String.Join("," , s1));
```

```
ISet<string> s2 = new SortedSet<string>();
s2.Add("Ferrara"); s2.Add("Ravenna"); s2.Add("Forlì");
Console.WriteLine(String.Join("," , s2));
```

Insieme ordinato

```
var s3 = new HashSet<string>();
s3.Add("Piacenza"); s3.Add("Cesena"); s3.Add("Reggio");
Console.WriteLine(String.Join("," , s3));
```

Type inference

```
Bologna,Modena,Parma
Ferrara,Forlì,Ravenna
Piacenza,Cesena,Reggio
```

Insieme ordinato

COSTRUZIONE DI COLLEZIONI: ESEMPI IN C#

- Costruzione di mappe (modificabili)

```
IDictionary<string,int> m = new Dictionary<string,int>();
m.Add("Bologna", 395416);
m.Add("Modena", 189013);
m.Add("Parma", 200455);
Console.WriteLine(String.Join("," , m));
```

C#

```
IDictionary<string,int> m2 = new SortedDictionary<string,int>();
...
```

Mappa ordinata

```
var m3 = new Dictionary<string,int>();
...
```

Type inference

```
[Modena, 189013],[Parma, 200455],[Bologna, 395416]
[Bologna, 395416],[Modena, 189013],[Parma, 200455]
[Piacenza, 118260],[Cesena, 97465],[Reggio, 171084]
```

Ordinata

COSTRUZIONE DI COLLEZIONI: ESEMPI IN C#

- Costruzione di mappe immodificabili
  - la mappa si crea solo inizialmente vuota, con la costante **`Empty`**
  - successive «modifiche» producono via via nuove mappe

```
var map1 = ImmutableSortedDictionary<String,int>.Empty;
var map2 = map1.Add("Topolinia", 1400)
               .Add("Paperopoli", 260);
Console.WriteLine("Map1: " + String.Join(", ", map1));
Console.WriteLine("Map2: " + String.Join(", ", map2));
```

C#

```
Map1:
Map2: [Paperopoli, 260], [Topolinia, 1400]
```

COSTRUZIONE DI COLLEZIONI: ESEMPI IN SCALA

- Costruzione di liste modificabili (Buffer)

Scala

```
val l1: Buffer[String] = new ListBuffer[String]();
l1 +=  "Bologna"; l1 += "Modena"; l1 += "Parma";
println(l1);
```

alias per addOne

```
val l2: Buffer[String] = new ArrayBuffer[String]();
l2.addOne("Ferrara"); l2.addOne("Ravenna");
println(l2);
```

```
val l3 = new ListBuffer[String]();
l3 addOne "Piacenza"; l3 addOne "Cesena";
println(l3);
```

Type inference

Metodi unari usabili anche come operatori infissi

```
ListBuffer(Bologna, Modena, Parma)
ArrayBuffer(Ferrara, Ravenna)
ListBuffer(Piacenza, Cesena)
```

COSTRUZIONE DI COLLEZIONI: ESEMPI IN SCALA

- Costruzione di liste immodificabili (List)

Scala

```
val l1: scala.collection.immutable.List[String] =
                List("Bologna", "Modena", "Parma");
println(l1);
```

Metodo factory

Type inference

```
val l3 = List("Piacenza", "Cesena", "Reggio");
println(l3);
```

Metodo factory

```
List(Bologna, Modena, Parma)
List(Piacenza, Cesena, Reggio)
```

COSTRUZIONE DI COLLEZIONI: ESEMPI IN SCALA

- Costruzione di set modificabili

Scala

```
val s1: Set[String] = new HashSet[String]();
s1 +=  "Bologna"; s1 += "Modena"; s1 += "Parma";
println(s1);
```

Insieme ordinato

```
val s2: Set[String] = new TreeSet[String]();
s2.addOne("Ferrara"); s2.addOne("Ravenna");
println(s2);
```

Type inference

```
val s3 = new HashSet[String]();
s3 addOne "Piacenza"; s3 addOne "Cesena";
println(s3);
```

```
HashSet(Parma, Bologna, Modena)
TreeSet(Ferrara, Ravenna)
HashSet(Piacenza, Cesena)
```

Insieme ordinato

COSTRUZIONE DI COLLEZIONI: ESEMPI IN SCALA

- Costruzione di set immodificabili

Scala

```
val l1: Set[String] =
                  Set("Bologna", "Modena", "Parma");
println(l1);
```

Metodo factory

Type inference

Insieme ordinato

```
val l3 = SortedSet("Piacenza", "Cesena", "Reggio");
println(l3);
```

Metodo factory

```
Set(Bologna, Modena, Parma)
TreeSet(Cesena, Piacenza, Reggio)
```

Insieme ordinato

COSTRUZIONE DI COLLEZIONI: ESEMPI IN SCALA

- Costruzione di mappe modificabili

Scala

```
val s1: Map[String,Int] = new HashMap[String,Int]();
s1 +=  "Bologna" -> 395416;
s1 += "Modena" -> 189013;
s1 += "Parma" -> 200455;
println(m);
```

Le entry si esprimono nella forma **key -> value**

```
val s2: Map[String,Int] = new TreeMap[String,Int]();
...
```

Mappa ordinata

```
val s3 = new HashMap[String,Int]();
...
```

Type inference

```
HashMap(Parma -> 200455, Bologna -> 395416, Modena -> 189013)
TreeMap(Bologna -> 395416, Modena -> 189013, Parma -> 200455)
HashMap(Parma -> 200455, Bologna -> 395416, Modena -> 189013)
```

Ordinata

COSTRUZIONE DI COLLEZIONI: ESEMPI IN SCALA

- Costruzione di mappe immodificabili

Scala

```
val m1: Map[String,Int] = Map(
  "Bologna" -> 395416,
  "Modena" -> 189013,
  "Parma" -> 200455);
println(m1);
```

Metodo factory

Le entry si esprimono nella forma `key -> value`

```
val m2 = SortedMap(
  "Bologna" -> 395416,
  "Modena" -> 189013,
  "Parma" -> 200455););
println(m2);
```

Insieme ordinato

Type inference

Metodo factory

```
Map(Bologna -> 395416, Modena -> 189013, Parma -> 200455)
TreeMap(Bologna -> 395416, Modena -> 189013, Parma -> 200455)
```

Mappa ordinata

COSTRUZIONE DI COLLEZIONI: ESEMPI IN KOTLIN

- Costruzione di liste modificabili

Kotlin

```
val l1: MutableList<String> = ArrayList<String>();
l1 +=  "Bologna"; l1 += "Modena"; l1 += "Parma";
println(l1);
```

alias per add

```
val l2: MutableList<String> = ArrayList<String>();
l2.add("Ferrara"); l2.add("Ravenna"); l2.add("Forlì");
println(l2);
```

```
val l3 = ArrayList<String>();
l3.add("Piacenza"); l3.add("Cesena"); l3.add("Reggio");
println(l3);
```

Type inference

```
[Bologna, Modena, Parma]
[Ferrara, Ravenna, Forlì]
[Piacenza, Cesena, Reggio]
```

COSTRUZIONE DI COLLEZIONI: ESEMPI IN KOTLIN

- Costruzione di liste immodificabili

Kotlin

```
val l1: List<String> =
                listOf("Bologna", "Modena", "Parma");
println(l1);
```

Metodo factory

Type inference

```
val l3 = listOf("Piacenza", "Cesena", "Reggio");
println(l3);
```

Metodo factory

```
[Bologna, Modena, Parma]
[Piacenza, Cesena, Reggio]
```

COSTRUZIONE DI COLLEZIONI: ESEMPI IN KOTLIN

- Costruzione di set modificabili

Kotlin

```
val s1: MutableSet<String> = HashSet<String>();
s1 +=  "Bologna"; s1 += "Modena"; s1 += "Parma";
println(s1);
```

```
val s2: MutableSet<String> = LinkedHashSet<String>();
s2.add("Ferrara"); s2.add("Ravenna"); s2.a      "Forlì");
println(s2);
```

Non è « ordinato» ma mantiene l'ordine di inserimento

```
val s3 = HashSet<String>();
s3.add("Piacenza"); s3.add("Cesena"); s3.add("Reggio");
println(s3);
```

Type inference

```
[Parma, Bologna, Modena]
[Ferrara, Ravenna, Forlì]
[Reggio, Piacenza, Cesena]
```

Ordine di inserimento

COSTRUZIONE DI COLLEZIONI: ESEMPI IN KOTLIN

- Costruzione di set immodificabili

Kotlin

```
val s1: Set<String> =
                setOf("Bologna", "Modena", "Parma");
println(s1);
```

Metodo factory

Mantiene l'ordine di inserimento

Type inference

```
val s3 = setOf("Piacenza", "Cesena", "Reggio");
println(s3);
```

Metodo factory

Mantiene l'ordine di inserimento

```
[Bologna, Modena, Parma]
[Piacenza, Cesena, Reggio]
```

Ordine di inserimento

COSTRUZIONE DI COLLEZIONI: ESEMPI IN KOTLIN

- Costruzione di mappe modificabili

```kotlin
val m1: MutableMap<String,Int> = LinkedHashMap<String,Int>();
m1["Bologna"] = 395416;
m1["Modena"] = 189013;
m1["Parma"] = 200455;
println(m1);
```

Kotlin

Le entry si possono assegnare anche con la notazione **`map[key] = value`**

```kotlin
val m2: MutableMap<String,Int> = LinkedHashMap<String,Int>();
m2.put("Bologna", 395416);
m2.put("Modena", 189013);
m2.put("Parma" , 200455);
println(m2);
```

Kotlin

Altrimenti si assegnano normalmente con metodo **`put`**

```
{Bologna=395416, Modena=189013, Parma=200455}
{Bologna=395416, Modena=189013, Parma=200455}
```

Ordinata

COSTRUZIONE DI COLLEZIONI: ESEMPI IN KOTLIN

- Costruzione di mappe modificabili

```
val m3 = HashMap<String,Int>();
m3.put("Modena", 189013);
m3.put("Parma", 200455);
val m4 = m3.plus(Pair("Bologna",395416));
println(m4);
println(m3); // ma s3 è rimasta intoccata
```

Type inference

Kotlin

Il metodo **plus** aggiunge una entry (**Pair**) a una *nuova* mappa restituita per l'occasione

```
{Parma=200455, Modena=189013, Bologna=395416}
{Parma=200455, Modena=189013}
```

m4

m3

COSTRUZIONE DI COLLEZIONI: ESEMPI IN KOTLIN

- Costruzione di mappe immodificabili

Kotlin

```
val m1: Map<String,Int> = mapOf(
  "Bologna" to 395416,
  "Modena" to 189013,
  "Parma" to 200455);
println(m1);
```

Metodo factory

Le entry si esprimono nella forma *key to value*

```
val m2 = mapOf(
  "Bologna" to 395416,
  "Modena" to 189013,
  "Parma" to 200455);
println(m2);
```

Type inference

Metodo factory

```
{Bologna=395416, Modena=189013, Parma=200455}
{Bologna=395416, Modena=189013, Parma=200455}
```

RIPRENDENDO GLI ESEMPI: Java

Java

Nell'esercizio n. 1 (**`Set`**) si può scegliere fra:

- **`HashSet`**: insieme non ordinato, tempo d'accesso costante
- **`TreeSet`**: insieme ordinato, tempo di accesso non costante

Output con `HashSet`:

```
>java FindDups Io sono Io esisto Io parlo
Parola duplicata: Io
Parola duplicata: Io
4 parole distinte: [Io, parlo, esisto, sono]
```

ordine qualunque

Output con `TreeSet`:

```
>java FindDups Io sono Io esisto Io parlo
Parola duplicata: Io
Parola duplicata: Io
4 parole distinte: [Io, esisto, parlo, sono]
```

ordine alfabetico!

RIPRENDENDO GLI ESEMPI: Java

Negli esercizi n. 2 e 3 (**`List`**) si può scegliere fra:

Java

- **`ArrayList`**: i principali metodi eseguono in tempo costante, mentre gli altri eseguono in un tempo lineare, ma con una costante di proporzionalità molto più bassa di **`LinkedList`**.
- **`LinkedList`**: il tempo di esecuzione è quello di una tipica realizzazione basata su puntatori; implementa anche le interfacce **`Queue`** e **`Deque`**, offrendo così una coda FIFO
- `Vector`: versione reingegnerizzata e sincronizzata di **`ArrayList`**

L'output ovviamente *non varia* al variare dell'implementazione, in ossequio sia al concetto di lista come *sequenza* di elementi, sia alla semantica di **`add`** come *append.*

RIPRENDENDO GLI ESEMPI: Java

Java

Negli esercizi n. 4 e 5 (**`Map`**) si può scegliere fra:

- **`HashMap`**: tabella non ordinata, tempo d'accesso costante
- **`TreeMap`**: tabella ordinata, tempo di accesso non costante

Output con `HashMap`:

```
>java ContaFrequenza cane gatto cane pesce gatto gatto cane
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

Output con `TreeMap` *(elenco ordinato)*:

```
>java ContaFrequenzaOrd cane gatto cane pesce gatto gatto cane
3 parole distinte: {cane=3, gatto=3, pesce=1}
```

RIPRENDENDO GLI ESEMPI: C#

C#

```
ISet<string> s = new HashSet<string>();
for (int i=0; i<args.Length; i++)
 if (!s.Add(args[i]))
    Console.WriteLine("Parola duplicata: " + args[i]);
Console.WriteLine(s.Count + " parole distinte: " + String.Join(",", s));
```

Costruzione di strutture dati inizialmente vuote

```
    IList<string> list = new List<string>();
    for (int i=0; i
    Console.WriteLi
    swap(list, 2, 3
    Console.WriteLi

}

// ----------------

static void swap<T>
    T temp = list[i
    list[i] = list[
    list[j] = temp;
}
```

```
    IDictionary<string,int> m = new Dictionary<string,int>();
    for (int i=0; i<args.Length; i++) {
     int freq = m.ContainsKey(args[i]) ? m[args[i]] : 0;
     m[args[i]] = freq+1;
    }
    Console.WriteLine(m.Count.ToString() + " parole distinte:");
    Console.WriteLine(String.Join(",", m));           // stampa esattamente come sotto
    MyPrint(m);
    Console.WriteLine(String.Join(",", entrySet(m))); // stampa esattamente come sopra
}

public static void MyPrint(IDictionary<string,int> m){
    ICollection<string> keys = m.Keys;
    foreach (string key in keys) {
        Console.WriteLine(key + "\t" + m[key]);
    }
}

public static ISet<KeyValuePair<string,int>> entrySet(IDictionary<string,int> m) {
// simula la entrySet di Java
    ISet<KeyValuePair<string,int>> result = new HashSet<KeyValuePair<string,int>>();
    foreach(KeyValuePair<string,int> entry in m) result.Add(entry);
    return result;
}
```

RIPRENDENDO GLI ESEMPI: C#

- Riguardo all'equivalente di **`SortedMap`**
  - in C# non esiste un'interfaccia del tipo «**`ISortedDictionary`**»: si mantiene **`IDictionary`**, scegliendo implementazioni *sorted*
  - le implementazioni fra cui scegliere sono **`SortedDictionary`** e **`SortedList`**
    - **`SortedList`** è una struttura «duale», accessibile sia tramite indice (come una lista) sia tramite chiave (come una mappa)
    - rispetto a **`SortedDictionary`**, **`SortedList`** usa meno memoria; tuttavia **`SortedDictionary`** ha operazioni di inserimento e rimozione più veloci per i dati non ordinati - O(log n) - contro O(n) in **`SortedList`**; d'altra parte, se la mappa viene popolata in una sola volta da dati preventivamente ordinati, **`SortedList`** è più veloce

```
IDictionary<string,int> m = new SortedDictionary<string,int>();
// <-- SOSTITUITO QUI Dictionary con SortedDictionary

IDictionary<string,int> m = new SortedList<string,int>();
// <-- SOSTITUITO QUI Dictionary con SortedList
```

C#

RIPRENDENDO GLI ESEMPI: Scala

Scala

```
val s: Set[String] = new HashSet();
for (i <- 0 until arg.size)
 if (!s.contains(arg(i)))
    s += arg(i);
 else
    println(
println(s.si
```

```
    var list : Buffer[String] = new ListBuffer[String]();

    for (i <- 0 until arg.length) list += arg(i);
    println(list);
}

//

def

}
```

Costruzione di strutture dati inizialmente vuote

```
    val m : Map[String,Int] = new HashMap[String,Int]();

    for (i <- 0 until arg.length) {
         val freq : Option[Int] = m.get(arg(i));
         m += arg(i)->(freq.getOrElse(0)+1);
    }
    println(m.size.toString() + " parole distinte:");
    println(m);
    //
    myPrint(m);
    myPrintEntries(m);
    //
    println(entrySet(m))
}

def myPrint(m : Map[String,Int]) : Unit = { // stampa "chiave \t valore"
    val keys: scala.collection.Set[String] = m.keySet;
    for (key <- keys) {
        println(key + "\t" + m.get(key).get);
    }
}
```

RIPRENDENDO GLI ESEMPI: Kotlin

Kotlin

```
val s: MutableSet<String> = HashSet();
for (i in 0..args.size-1)
 if (!s.add(args[i]))
    println("Parola duplicata: " + args[i]);
println(s.size.toString() + " parole distinte: "+s);
```

Costruzione di strutture dati inizialmente vuote

```
        var list : MutableList<String> = ArrayList<String>();

        for (i in 0..arg.size-1) list.add(arg[i]);
        println(list);
        swap(list, 2, 3);
        println(list);
}

fun <T> swap(list : MutableList<T>, i:Int, j:Int) : Unit {
    val temp = list.get(i);
    list.set(i, list.get(j));
    list.set(j, temp);
}
```

«Conversioni» fra strutture dati

"CONVERSIONI" FRA STRUTTURE

- A volte occorre "convertire" un tipo di struttura dati in un altro
  - obiettivo: *produrre una nuova struttura dati, di diversa tipologia,* contenente gli stessi elementi (es. List → Set, List → Array, etc.)
- A tal fine, ogni linguaggio adotta tipicamente un mix fra questi approcci:
  - le classi-collection definiscono un costruttore o un metodo factory statico che accetta come argomento *un'altra* ***`Collection`***
  - le collection definiscono metodi ***`toSomething`*** che producono una nuova struttura di tipo ***`Something`***
  - gli array, esterni alla tassonomia collection, richiedono metodi ad hoc
  - in ogni caso, la nuova struttura prodotta è un *duplicato* dell'originale con copia *shallow*

"CONVERSIONI" FRA STRUTTURE

- In Java

  Java

  – tutte le classi-collection definiscono un costruttore che accetta come argomento *un'altra* ***`Collection`***

  – l'interfaccia-base ***`Collection`*** contiene alcuni metodi ***`toArray`*** per convertire una qualunque collection in un array

  – fra liste e array sono definiti metodi di conversione rapidi:
  → da array a lista: metodo statico ***`Arrays.asList(array)`***
  → da lista ad array: metodo di **`List`** ***`toArray(tipo)`***

- In Kotlin

  Kotlin

  – sono definiti metodi di conversione ***`toSomething`*** per tutti i casi possibili (es. **`toSet`**, **`toSortedSet`**, **`toMutableList`**, etc.)

  → in particolare, da array a lista: metodo ***`toList()`***
  → in particolare, da lista ad array: metodo ***`toTypedArray()`***

"CONVERSIONI" FRA STRUTTURE

- In Scala

Scala

  - le classi-collection definiscono un metodo statico ***`from`*** che accetta come argomento *un'altra* ***`Collection`***
  - l'interfaccia-base ***`Iterable`*** dichiara metodi ***`toSomething`*** senza argomenti per tutti i casi possibili (es. **`toSet`**, **`toList`**, **`toSeq`**, etc.)

    → in particolare, da array a lista: metodo ***`toList`***
    → in particolare, da lista ad array: metodo ***`toArray[tipo]`***

- In C#

C#

  - come in Java, le classi-collection definiscono un costruttore che accetta come argomento *un'altra* ***`Collection`***
  - la libreria **`System.Linq`** aggiunge quattro metodi di conversione, richiamabili poi sui normali tipi collection: **`toArray`**, **`toHashSet`**, **`toList`**, **`toDictionary`** (quest'ultimo richiede lambda expression)

"CONVERSIONI" FRA STRUTTURE: JAVA

Esempio 1: da lista a set

Java

```
List<String> l1 =  List.of("Pippo", "Pluto", "QuiQuoQua",
 "Paperino", "Zio Paperone");          System.out.println(l1);
Set<String>  s1 = new HashSet<>(l1);   System.out.println(s1);
Set<String>  s2 = new TreeSet<>(l1);   System.out.println(s2);
```

```
[Pippo, Pluto, QuiQuoQua, Paperino, Zio Paperone]
[Paperino, QuiQuoQua, Pippo, Pluto, Zio Paperone]
[Paperino, Pippo, Pluto, QuiQuoQua, Zio Paperone]
```

Esempio 2: da lista immodificabile a lista modificabile

Java

```
List<String> l1 =  List.of("Pippo", "Pluto", "QuiQuoQua",
 "Paperino", "Zio Paperone");          System.out.println(l1);
List<String> l2 = new ArrayList<>(l1);   System.out.println(l2);
l2.add("Archimede"); l2.add("Paperino"); System.out.println(l2);
```

```
[Pippo, Pluto, QuiQuoQua, Paperino, Zio Paperone]
[Pippo, Pluto, QuiQuoQua, Paperino, Zio Paperone]
[Pippo, Pluto, QuiQuoQua, Paperino, Zio Paperone, Archimede, Paperino]
```

"CONVERSIONI" FRA STRUTTURE: JAVA

Esempio 3: modifiche ai set (verifica copia shallow)

Java

```
List<String> l1 =  List.of("Pippo", "Pluto", "QuiQuoQua",
 "Paperino", "Zio Paperone");          System.out.println(l1);
Set<String>  s1 = new HashSet<>(l1);   System.out.println(s1);
Set<String>  s2 = new TreeSet<>(l1);   System.out.println(s2);
s1.add("Paperoga");  System.out.println(s1);
s2.add("PaperUga");  System.out.println(s2);
```

```
[Pippo, Pluto, QuiQuoQua, Paperino, Zio Paperone]
[Paperino, QuiQuoQua, Pippo, Pluto, Zio Paperone]
[Paperino, Pippo, Pluto, QuiQuoQua, Zio Paperone]
[Paperino, Paperoga, QuiQuoQua, Pippo, Pluto, Zio Paperone]
[PaperUga, Paperino, Pippo, Pluto, QuiQuoQua, Zio Paperone]
```

"CONVERSIONI" FRA STRUTTURE: JAVA

Esempio 4: da set non ordinato a set ordinato

Set non ordinato

```
Set<String> keys = m.keySet();
for (String key: keys) System.out.println(key + "\t" + m.get(key));
```

Java

```
pesce    1
cane     3
gatto    3
```

Set ordinato

```
SortedSet<String> keys = new TreeSet<>(m.keySet());
for (String key: keys) System.out.println(key + "\t" + m.get(key));
```

Java

```
cane     3
gatto    3
pesce    1
```

"CONVERSIONI" FRA STRUTTURE: JAVA

Da lista ad array:

```
List<String> l1 =  List.of("Pippo", "Pluto", "QuiQuoQua",
 "Paperino", "Zio Paperone");          System.out.println(l1);
String[] a1 = l1.toArray(new String[0]);
```

Java

Falso argomento per specificare il tipo dell'array da produrre

```
[Pippo, Pluto, QuiQuoQua, Paperino, Zio Paperone]
{ "Pippo", "Pluto", "QuiQuoQua", "Paperino", "Zio Paperone" }
```

Da array a lista:

```
String[] arr = { "One", "Three", "Ten"};
System.out.println(String.join(",", arr);
List<String> list = Arrays.asList(arr);
```

Java

Trucchetto utile ***solo*** per array o liste di stringhe

Metodo statico della libreria Arrays

```
{ "One", "Three", "Ten" }
[One, Three, Ten]
```

"CONVERSIONI" FRA STRUTTURE: C#

C#

```
using System;
using System.Collections.Generic;
using System.Linq; // per ToList, ToArray, ToHashSet, ToDictionary

public class Program
{
    public static void Main(){

        // da array a lista
        string[] arr1 = {"cane", "gatto", "pappagallo", "canarino", "pescerosso"};
        IList<string> list1 = arr1.ToList();
        Console.WriteLine("{" + String.Join(",", list1) + "}");

        // da lista a lista
        List<string> list3 = new List<string>(list1);
        Console.WriteLine("{" + String.Join(",", list3) + "}");

        // da lista ad array
        List<int> intList1 = new List<int> { 10, 20, 10, 34, 113 };
        Console.WriteLine("{" + String.Join(",", intList1) + "}");
        int[] intArray1 = intList1.ToArray();
        Console.WriteLine("[" + String.Join(",", intArray1) + "]");

        // da lista a set
        ISet<int> intSet1 = intList1.ToHashSet();
        Console.WriteLine("(" + String.Join(",", intSet1) + ")");

        // da set ad array
        int[] intArray2 = intSet1.ToArray();
        Console.WriteLine("[" + String.Join(",", intArray2) + "]");

    }
}
```

```
{cane,gatto,pappagallo,canarino,pescerosso}
{cane,gatto,pappagallo,canarino,pescerosso}
{10,20,10,34,113}
[10,20,10,34,113]
(10,20,34,113)
[10,20,34,113]
```

"CONVERSIONI" FRA STRUTTURE: SCALA

Scala

```
object ConversioniFraStrutture {
  def main(args: Array[String]) : Unit = {

    var sList = List("b", "a", "c");  println(sList)
    var iList = List(12, -3, 27, -7); println(iList)
    var msList = scala.collection.mutable.Buffer("a", "b", "c");  println(msList)
    var miList = scala.collection.mutable.Buffer(12, -3, 27, -7); println(miList)

    var sArray  : Array[String] = sList.toArray[String];
    var iArray  : Array[Int]    = iList.toArray[Int];
    var msArray : Array[String] = msList.toArray[String];
    var miArray : Array[Int]    = miList.toArray[Int];
    sArray(2) = "ddd";  println(sArray.mkString(","))

    var sRes   = sArray.toList;     println(sRes);
    var sSet   = sList.toSet;       println(sSet);
    var ssSet  = scala.collection.SortedSet.from(sSet);
    var sList2 = sList.toBuffer;
    sList2 += "eee";                println(sList2);
  }

}
```

```
---------- Scala run ----------
List(b, a, c)
List(12, -3, 27, -7)
ArrayBuffer(a, b, c)
ArrayBuffer(12, -3, 27, -7)
b,a,ddd
List(b, a, ddd)
Set(b, a, c)
TreeSet(a, b, c)
ArrayBuffer(b, a, c, eee)
```

"CONVERSIONI" FRA STRUTTURE: KOTLIN

```
fun main() {
    var sList = listOf("b", "a", "c");                          println(sList)
    var iList = listOf(12, -3, 27, -7);                         println(iList)
    var msList = mutableListOf("a", "b", "c");                  println(msList)
    var miList = mutableListOf(12, -3, 27, -7);                 println(miList)

    var sArray  : Array<String> = sList.toTypedArray();         println(sArray.joinToString(","))
    var iArray  : Array<Int>    = iList.toTypedArray();         println(iArray.joinToString(","))
    var msArray : Array<String> = msList.toTypedArray();        println(msArray.joinToString(","))
    var miArray : Array<Int>    = miList.toTypedArray();        println(miArray.joinToString(","))

    sArray[2] = "ddd";  println(sArray.joinToString(","))

    var sRes   = sArray.toList();       println(sRes);
    var sSet   = sList.toSet();         println(sSet);
    var ssSet  = sList.toSortedSet();   println(ssSet);
    var sList2 = sList.toMutableList(); sList2.add("eee");  println(sList2);

}
```

```
[b, a, c]
[12, -3, 27, -7]
[a, b, c]
[12, -3, 27, -7]
b,a,c
12,-3,27,-7
a,b,c
12,-3,27,-7
b,a,ddd
[b, a, ddd]
[b, a, c]
[a, b, c]
[b, a, c, eee]
```

Kotlin

Esercizio
Ordinamenti e ricerche in strutture dati

JAVA: ESERCIZIO DI ORDINAMENTO & RICERCA

Java

- Obiettivo: sperimentare le funzioni di ordinamento e ricerca nelle strutture dati

Classe di appoggio: una semplice **`Persona`**, molto basica

  - proprietà: solo *nome* e *cognome*
  - ipotesi: *ordinamento naturale* per cognome e in subordine per nome

- Cosa serve
  - possibilità di inizializzare array e liste di persone
  - per gli array è facile → notazione *array literal* `{…}`
  - per le liste immodificabili è facile → factory `List.of(…)`
  - per le liste modificabili è meno semplice → necessità di un qualche *workaround* (costruzione per copia)

UNA SEMPLICE Persona CONFRONTABILE

Java

```
class Persona implements Comparable<Persona> {
  private String nome, cognome;
  public Persona(String nome, String cognome) {
    this.nome = nome;  this.cognome = cognome;
  }
  public String nome() {return nome;}
  public String cognome() {return cognome;}
  public String toString() {return nome + " " + cognome;}
  public int compareTo(Persona that) {
    int confrontoCognomi = cognome.compareTo(that.cognome);
    return (confrontoCognomi!=0 ? confrontoCognomi :
            nome.compareTo(that.nome));
  }
}
```

ESERCIZIO 1: ORDINARE UNA LISTA DI PERSONE

Java

```
class NameSort1 {
 public static void main(String args[]) {
  Persona[] elencoPersone = {
      new Persona("Eugenio", "Bennato"),
      new Persona("Roberto", "Benigni"),
      new Persona("Edoardo", "Bennato"),
      new Persona("Bruno", "Vespa")
  };
  List<Persona> l = Arrays.asList(elencoPersone);
  Collections.sort(l);
  System.out.println(l);
 }
}
```

Dovendola ordinare, serve una lista *modificabile*

Opzione#1: partiamo da un array e lo convertiamo in lista

Ordinamento lista tramite metodo statico della libreria **Collections**

Se il cognome è uguale valuta il nome, come è giusto

```
>java NameSort1
[Roberto Benigni, Edoardo Bennato, Eugenio Bennato, Bruno Vespa]
```

ESERCIZIO 1: ORDINARE UNA LISTA DI PERSONE

Java

```
class NameSort2 {
 public static void main(String args[]) {
  List<Persona> l = Arrays.asList( new Persona[] {
      new Persona("Eugenio", "Bennato"),
      new Persona("Roberto", "Benigni"),
      new Persona("Edoardo", "Bennato"),
      new Persona("Bruno", "Vespa")
  } );
  Collections.sort(l);
  System.out.println(l);
 }
}
```

Le due fasi unite senza passare da un array temporaneo esplicito

NB: `List.of` non funzionerebbe

`UnsupportedOperationException`
`ImmutableCollections`

```
>java NameSort2
[Roberto Benigni, Edoardo Bennato, Eugenio Bennato, Bruno Vespa]
```

ESERCIZIO 1: ORDINARE UNA LISTA DI PERSONE

Java

```
class NameSort3 {
 public static void main(String args[]) {
  List<Persona> l = new ArrayList<>(List.of(
      new Persona("Eugenio", "Bennato"),
      new Persona("Roberto", "Benigni"),
      new Persona("Edoardo", "Bennato"),
      new Persona("Bruno", "Vespa")
  ));
  Collections.sort(l);
  System.out.println(l);
 }
}
```

Opzione#2: partiamo da una lista immodificabile e la *convertiamo per copia* in una modificabile

```
>java NameSort3
[Roberto Benigni, Edoardo Bennato, Eugenio Bennato, Bruno Vespa]
```

ESERCIZIO 2: ORDINARE UN ARRAY DI PERSONE

Java

```
class NameSort4 {
  public static void main(String args[]) {
    Persona[] persone = {
        new Persona("Eugenio", "Bennato"),
        new Persona("Roberto", "Benigni"),
        new Persona("Edoardo", "Bennato"),
        new Persona("Bruno", "Vespa")
    };
    Arrays.sort(persone);
    System.out.println(Arrays.toString(persone));
  }
}
```

Ordinamento array tramite metodo statico della libreria **Arrays**

ESERCIZIO 2 bis: ORDINARE UN ARRAY DI PERSONE

Java

```
class NameSort4 {
 public static void main(String args[]) {
  List<Persona> l = List.of(
      new Persona("Eugenio", "Bennato"),
      new Persona("Roberto", "Benigni"),
      new Persona("Edoardo", "Bennato"),
      new Persona("Bruno", "Vespa")
  );
  Persona[] persone = l.toArray(new Persona[0]);
  Arrays.sort(persone);

  System.out.println(Arrays.toString(persone));
 }
}
```

Se per caso i dati fossero già in una lista e fosse necessario trasporli in un array

Convertiamo la lista in array col metodo **toArray**

Ordinamento come prima, tramite il metodo statico della libreria **Arrays**

ESERCIZIO 3: CERCARE IN UNA LISTA DI PERSONE

Java

```
class Search1 {
  public static void main(String args[]) {
    List<Persona> l = new ArrayList<>(List.of(
        new Persona("Eugenio", "Bennato"),
        new Persona("Roberto", "Benigni"),
        new Persona("Edoardo", "Bennato"),
        new Persona("Bruno", "Vespa")
    ));
    Collections.sort(l);
    System.out.println(Collections.binarySearch(l,
        new Persona("Bruno", "Vespa")));
    System.out.println(Collections.binarySearch(l,
        new Persona("Bruno", "Ape")));
  }
}
```

Serve una lista *modificabile* perché la ricerca binaria richiede che la lista sia ordinata

Ordinamento

C'è @ posizione 3

Non c'è → -1

```
>java Search1
3
-1
```

ESERCIZIO 3 bis: CERCARE IN UNA LISTA DI PERSONE

Java

```
class Search1 {
  public static void main(String args[]) {
    List<Persona> l = List.of(
        new Persona("Edoardo", "Bennato"),
        new Persona("Eugenio", "Bennato"),
        new Persona("Roberto", "Benigni"),
        new Persona("Bruno", "Vespa")
    );
    System.out.println(Collections.binarySearch(l,
        new Persona("Bruno", "Vespa")));
    System.out.println(Collections.binarySearch(l,
        new Persona("Bruno", "Ape")));
  }
}
```

Se la lista è già ordinata, può essere *immodificabile*

(ordinamento rimosso)

C'è @ posizione 3

Non c'è → -1

```
>java Search1
3
-1
```

ESERCIZIO 4: CERCARE IN UNA MAPPA DI PERSONE

Java

Versione estesa di Persona con codice fiscale (ID univoco)

Mappa ordinata

```
class Search2 {
  public static void
    Map<String,PersonaCF> l = new TreeMap<>(Map.of(
        "BNNGNEyymddxxxxu",
        new PersonaCF("Eugenio", "Bennato","BNNGNEyymddxxxxu"),
        "BNGRRTyymddxxxxu",
        new PersonaCF("Roberto", "Benigni","BNGRRTyymddxxxxu"),
        "BNNDRDyymddxxxxu",
        new PersonaCF("Edoardo", "Bennato","BNNDRDyymddxxxxu"),
        "VSPBRNyymddxxxxu",
        new PersonaCF("Bruno", "Vespa","VSPBRNyymddxxxxu")
    ));
    System.out.println(l);
    System.out.println(l.get("BNGRRTyymddxxxxu"));
  }
}
```

```
>java Search2
{BNGRRTyymddxxxxu=Roberto Benigni, … }
Roberto Benigni
```

L'ESERCIZIO IN C#: ORDINARE UNA LISTA DI PERSONE

C#

```
class NameSort1 {
 public static void Main() {
    Persona[] elencoPersone = {
         new Persona("Eugenio", "Bennato"),
         new Persona("Roberto", "Benigni"),
         new Persona("Edoardo", "Bennato"),
         new Persona("Bruno", "Vespa")
    };
    List<Persona> list1 = new List<Persona>(elencoPersone);
    list1.Sort();
    Console.WriteLine(String.Join(", ", list1));
  }
}
```

```
List<Persona> list2 = new List<Persona> {
    new Persona("Eugenio", "Bennato"),
    new Persona("Roberto", "Benigni"),
    new Persona("Edoardo", "Bennato"),
    new Persona("Bruno", "Vespa")
};
```

Variante (esercizi #2-3-4)

```
Roberto Benigni, Edoardo Bennato, Eugenio Bennato, Bruno Vespa
```

L'ESERCIZIO IN C#: CERCARE IN UNA LISTA DI PERSONE

C#

```
class Search1 {
 public static void Main() {
    Persona[] elencoPersone = {
         new Persona("Eugenio", "Bennato"),
         new Persona("Roberto", "Benigni"),
         new Persona("Edoardo", "Bennato"),
         new Persona("Bruno", "Vespa")
    };
    List<Persona> list1 = new List<Persona>(elencoPersone);
    list1.Sort();
    Console.WriteLine(list1.BinarySearch(new Persona("Bruno", "Vespa")));
    Console.WriteLine(list1.BinarySearch(new Persona("Bruno", "Ape")));
 }
}
```

NB: il metodo Sort è definito solo nella classe List, non nell'interfaccia IList

```
3
-1
```

L'ESERCIZIO IN C#: CERCARE IN UNA MAPPA DI PERSONE

C#

```
class Search1 {
 public static void Main() {
    IDictionary<String,Persona> map = new SortedDictionary<String,Persona>{
         {"BNNGNEyymddxxxxu",
               new Persona("Eugenio", "Bennato","BNNGNEyymddxxxxu")},
         {"BNGRRTyymddxxxxu",
               new Persona("P                                  Tyymddxxxxu")},
         {"BNNDRDyymddxxxxu",
               new Persona("Edoardo", "Bennato","BNNDRDyymddxxxxu")},
         {"VSPBRNyymddxxxxu",
               new Persona("Bruno", "Vespa","VSPBRNyymddxxxxu")}
    };
    Console.WriteLine(String.Join(", ", map));
    Persona p;
    Console.WriteLine(map.TryGetValue("BNGRRTyymddxxxxu", out p));
    Console.WriteLine(p);
 }
}
```

Mappa ordinata

Versione estesa di Persona con codice fiscale (ID univoco)

Metodo di estrazione con risultato bool (successo/fallimento)

Argomento di uscita

```
[BNGRRTyymddxxxxu, Roberto Benigni], … ]
True
Roberto Benigni
```

L'ESERCIZIO IN SCALA: ORDINARE E CERCARE IN UNA LISTA

Scala

```
object SortAndSearch1 {
 def main(args: Array[String]) : Unit = {
    val elencoPersone : Array[Persona] = Array(
        new Persona("Eugenio", "Bennato"),
        new Persona("Roberto", "Benigni"),
        new Persona("Edoardo", "Bennato"),
        new Persona("Bruno", "Vespa")
    );
  val l = elencoPersone.toList;
  val l2 = l.sorted;
  println(l2);
  println(l.indexOf(new Persona("Bruno", "Vespa")));
  println(l.indexOf(new Persona("Bruno", "Ape")));
 }
}
```

```
val listaPersone = List(
    new Persona("Eugenio", "Bennato"),
    new Persona("Roberto", "Benigni"),
    new Persona("Edoardo", "Bennato"),
    ona("Bruno", "Vespa")
```

Lista immodificabile

Il metodo sorted produce una nuova lista ordinata

NB: la classe Persona deve essere completa di equals & hashCode

```
List(Roberto Benigni, Edoardo Bennato, Eugenio Bennato, Bruno Vespa)
3
-1
```

L'ESERCIZIO IN KOTLIN: CERCARE IN UNA MAPPA DI PERSONE

Scala

```
object SortAndSearch2 {
 def main(args: Array[String]) : Unit = {
    val map : Map[String,Persona] = Map(
        "BNNGNEyymddxxxxu" ->
             new Persona("Eugenio", "Bennato","BNNGNEyymddxxxxu"),
        "BNGRRTyymddxxxxu" ->
             new Persona("Roberto", "Benigni","BNGRRTyymddxxxxu"),
        "BNNDRDyymddxxxxu" ->
             new Persona("Edoardo", "Bennato","BNNDRDyymddxxxxu"),
        "VSPBRNyymddxxxxu" ->
             new Persona("Bruno", "Vespa","VSPBRNyymddxxxxu")
    );
    println(map);
    println(map.get("BNGRRTyymddxxxxu"));
 }
}
```

Mappa non ordinata

Versione estesa di Persona con codice fiscale (ID univoco)

```
{Map(BNNGNEyymddxxxxu -> Eugenio Bennato, …)
Some(Roberto Benigni)
```

L'ESERCIZIO IN KOTLIN: ORDINARE E CERCARE IN UNA LISTA

Kotlin

```
fun main() {
    val elencoPersone : Array<Persona> = arrayOf(
        Persona("Eugenio", "Bennato"),
        Persona("Roberto", "Benigni"),
        Persona("Edoardo", "Bennato"),
        Persona("Bruno", "Vespa")
    );
  val l = elencoPersone.toMutableList();
  l.sort(); // esiste solo per liste modificabili
  println(l);
  println(l.binarySearch(Persona("Bruno", "Vespa")));
  println(l.binarySearch(Persona("Bruno", "Ape")));
}
```

```
val listaPersone = mutableListOf(
    Persona("Eugenio", "Bennato"),
    Persona("Roberto", "Benigni"),
    Persona("Edoardo", "Bennato"),
    Persona("Bruno", "Vespa")
);
```

NB: la classe Persona deve essere completa di equals & hashCode

```
[Roberto Benigni, Edoardo Bennato, Eugenio Bennato, Bruno Vespa]
3
-1
```

L'ESERCIZIO IN KOTLIN: CERCARE IN UNA MAPPA DI PERSONE

Kotlin

```
fun main() {
  val map = mapOf(
   "BNNGNEyymddxxxxu" to
       Persona("Eugenio", "Bennato","BNNGNEyymddxxxxu"),
   "BNGRRTyymddxxxxu" to
       Persona("Roberto", "Benigni","BNGRRTyymddxxxxu"),
   "BNNDRDyymddxxxxu" to
       Persona("Edoardo", "Bennato","BNNDRDyymddxxxxu"),
   "VSPBRNyymddxxxxu" to
       Persona("Bruno", "Vespa","VSPBRNyymddxxxxu")
   );
   println(map);
   println(map.get("BNGRRTyymddxxxxu"));
}
```

Mappa non ordinata

Versione estesa di Persona con codice fiscale (ID univoco)

```
{BNNGNEyymddxxxxu=Eugenio Bennato, … }
Roberto Benigni
```

APPROFONDIMENTO FACOLTATIVO
Java: EnumSet & EnumMap

ULTERIORI IMPLEMENTAZIONI JAVA
`EnumSet & EnumMap`

Java

- **`EnumSet`** è un efficiente set per enumerativi
- A differenza delle altre implementazioni, questa è una classe astratta, *priva di costruttori pubblici*
- Le istanze si costruiscono tramite *metodi factory*
  - **`allOf(Class<E> elementType)`**
  - **`complementOf(EnumSet<E> s)`**
  - **`copyOf(Collection<E> c)`**
  - **`copyOf(EnumSet<E> s)`**
  - **`noneOf(Class<E> elementType)`**
  - cinque versioni di **`of(…)`**, da 1 a 5 argomenti
  - **`range(E from, E to)`**

Ciascuno di essi prima specializza la classe astratta *derivandone una concreta adatto al caso specifico*, poi la instanzia.

ESEMPIO: `EnumSet`

Java

- Creiamo un **`EnumSet`** di **`DayOfWeek`** e lavoriamoci un po':

```
jshell> Set<DayOfWeek> s = EnumSet.of(DayOfWeek.MONDAY)
s ==> [MONDAY]

jshell> s.add(DayOfWeek.FRIDAY)
$4 ==> true

jshell> s
s ==> [MONDAY, FRIDAY]
```

- Creazione di range (con o senza import static su **`DayOfWeek`**)

```
jshell> EnumSet.range(DayOfWeek.WEDNESDAY, DayOfWeek.SATURDAY)
$8 ==> [WEDNESDAY, THURSDAY, FRIDAY, SATURDAY]
```

```
jshell> import static java.time.DayOfWeek.*

jshell> EnumSet.range(MONDAY,WEDNESDAY)
$10 ==> [MONDAY, TUESDAY, WEDNESDAY]
```

- Creazione di un set inizialmente vuoto (occhio all'argomento!):

```
jshell> EnumSet<DayOfWeek> ss = EnumSet.noneOf(DayOfWeek.class)
ss ==> []
```

ESEMPIO: `EnumSet` (segue)

Java

- **`EnumSet`** non accetta oggetti nulli (altrimenti, NPE):

```
jshell> DayOfWeek d = null;
d ==> null

jshell> ss.add(d)
|  java.lang.NullPointerException thrown
|        at EnumSet.typeCheck (EnumSet.java:395)
|        at RegularEnumSet.add (RegularEnumSet.java:161)
|        at RegularEnumSet.add (RegularEnumSet.java:36)
|        at (#13:1)
```

- Ulteriori esempi d'uso:

```
jshell> EnumSet.complementOf(ss)
$15 ==> [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]

jshell> EnumSet.complementOf(EnumSet.range(MONDAY,WEDNESDAY))
$16 ==> [THURSDAY, FRIDAY, SATURDAY, SUNDAY]
```

```
jshell> for(DayOfWeek d: EnumSet.range(MONDAY,WEDNESDAY))
   ...> System.out.println(d)
MONDAY
TUESDAY
WEDNESDAY
```

ULTERIORI IMPLEMENTAZIONI Java
`EnumSet` & `EnumMap`

Java

- **`EnumMap`** è un'efficiente mappa per enumerativi
- A differenza di **`EnumSet`**, ha normali costruttori pubblici:
  - `EnumMap(Class<K> keyType)`
  - `EnumMap(EnumMap<K,? extends V> m)`
  - `EnumMap(Map<K,? extends V> m)`
- e offre i classici metodi delle mappe
  - `put`
  - `get`
  - `keySet`

Analogamente a **`EnumSet`**, anche questa aborrisce *chiavi* nulle

```
jshell> m.put(null, 0.5)
|  java.lang.NullPointerException thrown
|        at EnumMap.typeCheck (EnumMap.java:743)
|        at EnumMap.put (EnumMap.java:264)
|        at (#31:1)
```

ESEMPIO: `EnumMap`

- Creiamo una **`EnumMap`** di **`DayOfWeek`** inizialmente vuota, inseriamoci tre coppie (giorno, costo della sosta), e lavoriamoci un po':

```
jshell> EnumMap<DayOfWeek,Double> m = new EnumMap<>(DayOfWeek.class)
m ==> {}

jshell> m.put(TUESDAY, 0.50)
$22 ==> null

jshell> m
m ==> {TUESDAY=0.5}

jshell> m.put(FRIDAY, 1.10)
$24 ==> null

jshell> m.put(SATURDAY, 1.50)
$25 ==> null

jshell> m
m ==> {TUESDAY=0.5, FRIDAY=1.1, SATURDAY=1.5}

jshell> m.put(FRIDAY, 1.20)
$27 ==> 1.1

jshell> m
m ==> {TUESDAY=0.5, FRIDAY=1.2, SATURDAY=1.5}
```

Il metodo put restituisce il *precedente valore* associato a quella chiave (*null* se non c'era)

Java