Programmazione Orientata agli Oggetti

Collezioni:
Insiemi generici

Sommario

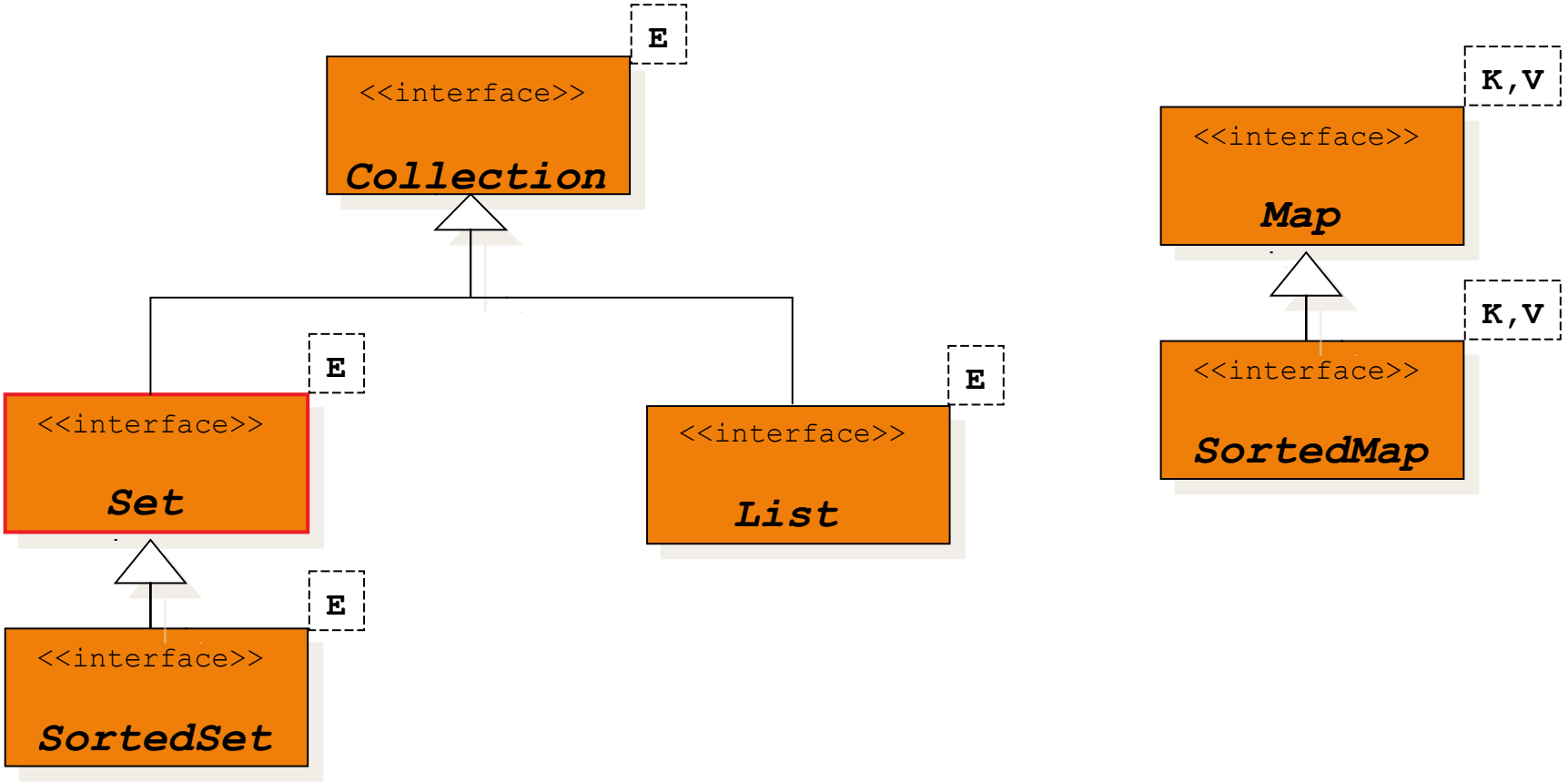
- L'interface **`Set<E>`**
- Criteri equivalenza tra elementi
- **`HashSet<E>`**
  - Tavole Hash (intuizione)
  - **`hashCode()`**, **`equals()`**
- **`TreeSet<E>`**
  - **`compare()`**, **`compareTo()`**

Sommario

- L'interface **`Set<E>`**
- Criteri equivalenza tra elementi
- **`HashSet<E>`**
  - Tavole Hash (intuizione)
  - **`hashCode()`, `equals()`**
- **`TreeSet<E>`**
  - **`compare()`, `compareTo()`**

Collezioni: Principali Interface

- Le principali *interface* del package **java.util**



- Per ognuna di queste interface il package offre diverse implementazioni

Insiemi: Interface *Set<E>*

- Un insieme (set) è una collezione *che non contiene duplicati*
- L'interface **`Set<E>`**, che estende l'interface **`Collection<E>`**, offre tutti e soli i metodi della interface **`Collection<E>`**, con la restrizione (riportata nel contratto) che le classi che la implementano si impegnano a non ammettere la presenza di elementi *duplicati*
- E’ necessario stabilire un criterio di equivalenza tra gli elementi dell’insieme al fine di rilevare (e gestire) l'inserimento di duplicati

Sommario

- L'interface **`Set<E>`**
- Criteri equivalenza tra elementi
- Tavole Hash (intuizione)
- **`HashSet<E>`**
  - **`hashCode(), equals()`**
- **`TreeSet<E>`**
  - **`compare(), compareTo()`**

Criterio di Equivalenza tra Elementi di un Insieme (1)

- Nel Java Collection Framework abbiamo due implementazioni di **`Set<E>`**, rispettivamente **`SortedSet<E>`**:
  - **`HashSet<E>`**
  - **`TreeSet<E>`**
- Attenzione: nelle due classi concrete il criterio di equivalenza tra elementi non è definito nella stessa maniera:
  - **`HashSet<E>`**
    si basa sui metodi **`equals()`** e **`hashCode()`**
  - **`TreeSet<E>`**
    viene «indotto» dal criterio di ordinamento (>>)

Criterio di Equivalenza tra Elementi di un Insieme (2)

- Per usare correttamente le implementazioni di **`Set<E>`** è necessario che le classi degli oggetti destinati a svolgere il ruolo di elementi dell'insieme, definiscano il criterio di equivalenza oggetti/elementi desiderato
- **`HashSet<E>`** richiede che tali classi ridefiniscano opportunamente due metodi (ereditati da **`Object`**) che servono a verificare ed evitare la presenza di duplicati
  - il metodo *equals()*: **`public boolean equals(Object o)`**
  - **ed anche** il metodo *hashCode()*: **`public int hashCode()`**
- **`TreeSet<E>`** richiede
  - che la classe degli elementi definisca un ordinamento naturale (quindi che implementi **`Comparable<E>`**)
  - *oppure*, che al momento della costruzione dell'insieme venga passato un comparatore (**`Comparator<E>`**) che definisca una relazione d'ordine per gli oggetti di tipo **`E`**

Insiemi: Criteri di Equivalenza

- Esempio: se sono interessato ad usare un insieme di oggetti **`Persona`** (**`Set<Persona>`**):
  - Se decidiamo di usare l'implementazione **`HashSet`**, la classe **`Persona`** deve:
    - ridefinire il metodo *equals()*
      `public boolean equals(Object that)`
    - *ed anche* ridefinire il metodo *hashCode()*
      `public int hashCode()`
  - Se decidiamo di usare l'implementazione **`TreeSet`**:
    - la classe **`Persona`** deve implementare l'interface **`Comparable<Persona>`**
    - *oppure*, l'oggetto **`TreeSet`** deve essere costruito passando un oggetto **`Comparator<Persona>`**

Sommario

- L'interface **`Set<E>`**
- Criteri equivalenza tra elementi
- **`HashSet<E>`**
  - Tavole Hash (intuizione)
  - **`hashCode()`**, **`equals()`**
- **`TreeSet<E>`**
  - **`compare()`**, **`compareTo()`**

Insiemi: Implementazione `HashSet<E>`

- Per comprendere appieno le motivazioni alla base di questa scelta progettuale è necessario discutere alcuni dettagli relativi all'implementazione
- L'implementazione di **`HashSet<E>`** si basa su una struttura dati fondamentale: *Tavola Hash*
- Ricapitoliamo velocemente alcuni aspetti (per una trattazione più approfondita, vedi il corso di *Algoritmi e strutture dati*)

Insiemi: Implementazione con Tavole Hash

- Una *funzione di hash* è una funzione che calcola un numero intero, detto *codice hash*, a partire dai dati di un oggetto, di modo che molto probabilmente (anche se non certamente) oggetti *non equivalenti* abbiano codici diversi
- Se invece due oggetti equivalenti finiscono per avere lo stesso codice di hash, si crea una situazione indesiderata
  - tale situazione è denominata *collisione*
- ✓ Una buona funzione di hash deve infatti minimizzare le collisioni per migliorare le prestazioni complessive

Codici *Hash*

- Esempi di codici hash per una stringa
  - Il numero di caratteri che compongono la stringa
    - "Pippo": 5
    - "Pluto": 5
    - "Paperino": 8
    - Non è un buon codice hash: troppe collisioni
  - La somma dei codici ASCII dei caratteri che compongono la stringa
    - "Pippo": 80+73+80+80+79=392
    - "Pluto": 80+76+85+84+79=400
    - "Paperino": 80+65+80+69+82+73+78+79=626
    - ... meglio ...
- Osservazione fondamentale: se i codici hash sono diversi allora le stringhe non sono eguali; il contrario non è necessariamente vero

Tavole Hash (1)

- Le funzioni di hash vengono usate per creare collezioni (mappe e insiemi) molto efficienti

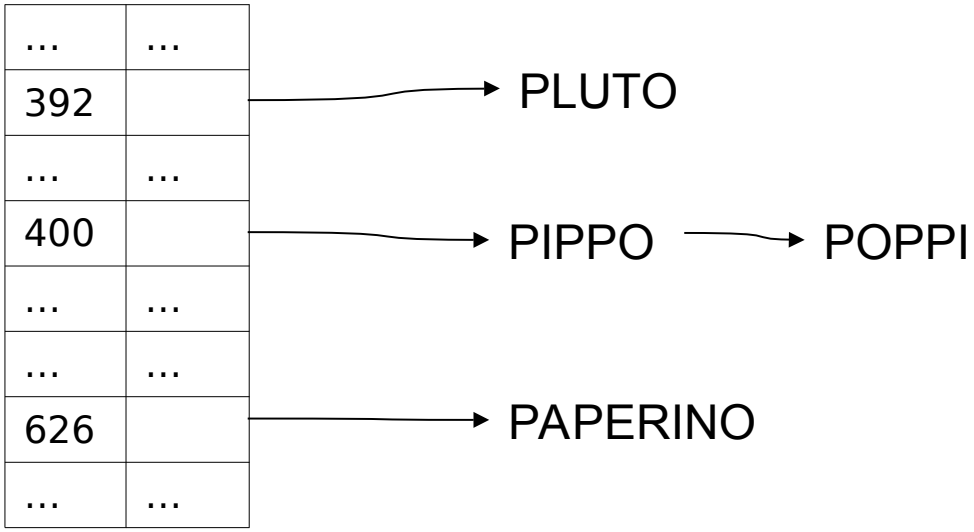
- ✓ L'idea è quella di usare rappresentazioni basate su array da indicizzare mediante i codici hash degli elementi

…	…
392	PLUTO
…	…
400	PIPPO
…	…
…	…
626	PAPERINO
…	…

Tavole Hash (2)

- Questa semplice idea però ha due problemi:
  - La dimensione dell'array
    - ✓ Se non basta a contenere tutti i possibili codici hash?
  - Le collisioni
    - ✓ Se trovo il posto già occupato da un altro elemento?
- Per ovviare al primo problema si deve scegliere una funzione di hash che generi codici all’interno di una gamma di valori ragionevolmente limitata
- Per ovviare al secondo problema si usano liste concatenate (vedi corso ASD)

Tavole Hash (3)



- Se la funzione di hash è ben definita (range di valori contenuto, poche collisioni) le operazioni di verifica di appartenenza, rimozione e inserimento in una collezione posso essere realizzate a costo costante

Tavole Hash (4)

- Le implementazioni degli insiemi (e delle mappe, vedremo>>) di basano sulle tavole hash
- In Java, il codice hash di un oggetto deve essere restituito dal metodo **`public int hashCode()`**
- Se vogliamo usare l'implementazione **`HashSet<E>`** di **`Set<E>`**, gli oggetti della collezione devono avere sia il metodo **`hashCode()`**, sia il metodo **`equals()`** opportunamente e coerentemente definiti
  - ✓ se non li definiamo, questi metodi vengono ereditati da **`java.lang.Object`** secondo una semantica che, di solito, *non* è quella desiderata: **`hashCode()`** restituisce [un numero che dipende dal]l'indirizzo in memoria dell'oggetto, **`equals()`** realizza una semantica equivalente a quella dell'operatore == per il confronto di riferimenti, e *non* dei contenuti degli oggetti referenziati

Sommario

- L'interface **`Set<E>`**
- Criteri equivalenza tra elementi
- **`HashSet<E>`**
  - Tavole Hash (intuizione)
  - **`hashCode(), equals()`**
- **`TreeSet<E>`**
  - **`compare(), compareTo()`**

Eguaglianza: il Metodo `equals()`

- Per ogni classe va definito il metodo

  `boolean equals(Object o)`
- Deve soddisfare le seguenti proprietà per qualsiasi riferimento non nullo **`x`**, **`y`**, **`z`**:
  - *Riflessività:* **`x.equals(x)`** deve restituire **`true`**
  - *Simmetria:* **`x.equals(y)`** deve restituire **`true`** se e solo se **`y.equals(x)`** restituisce **`true`**
  - *Transitività:* se **`x.equals(y)`** restituisce **`true`** e **`y.equals(z)`** restituisce **`true`**, allora **`x.equals(z)`** deve restituire **`true`**
  - **`x.equals(null)`** deve restituire **`false`**
  - Se due riferimenti **`x`** e **`y`** sono identici (**`x==y`**), allora **`x.equals(y)`** deve restituire **`true`**

Eguaglianza: il Metodo `hashCode()`

- Ogni volta che in una classe definiamo il metodo **`boolean equals(Object o)`**, **dobbiamo** definire anche il metodo **`int hashCode()`**
- Il metodo **`int hashCode()`** può essere definito come il calcolo di una semplice combinazione lineare (ad es. somma) dei codici hash delle variabili usate per verificare l'uguaglianza
  - tutte le classi della libreria standard hanno una implementazione ("naturalmente" definita) del metodo **`hashCode()`**
  - N.B.: se le variabili sono tipi primitivi numerici possiamo usare direttamente il loro valore (al limite convertendolo prima in un **`int`**)

Esempio

```
public class Calciatore {
    private String nome;
    private String cognome;
    private int numeroMaglia;
    … // metodi getter omessi
    @Override
    public boolean equals(Object o) {
          Calciatore that = (Calciatore)o;
          return this.getNome().equals(that.getNome()) &&
                 this.getCognome().equals(that.getCognome()) &&
              this.getNumeroMaglia()== that.getNumeroMaglia();
    }
    @Override
    public int hashCode() {
       return  this.getNome().hashCode()
             + this.getCognome().hashCode()
             + this.getNumeroMaglia();
    }
}
```

Gestione Duplicati in `HashSet<E>`

- In sintesi, per verificare la presenza di duplicati, **`HashSet<E>`** usa **`equals(Object o)`** *in combinazione* con il valore restituito dal metodo **`hashCode()`**
- In particolare l'implementazione di **`HashSet`**, per verificare se un elemento – *e* – è già presente nell'insieme:
  - prima, effettua una verifica molto *efficiente:* nell'insieme esistono già elementi con lo stesso codice hash di *e* ???
  - quindi, ma solo in caso affermativo (*collisione*) si verifica l'effettiva uguaglianza degli oggetti invocando il metodo **`equals(Object o)`**
    - ✔ per una valutazione «completa» e risolutiva dell'equivalenza, sebbene meno efficiente rispetto al metodo **`hashCode()`**

Due Metodi al Posto di Uno (1)

- Quindi i due metodi **`equals()`** ed **`hashCode()`** sono assolutamente *dipendenti* l'uno dall'altro e devono essere definiti in maniera coerente
- Perché costringere i programmatori utilizzatori a dover conoscere questi dettagli, non bastava il metodo **`equals()`**?
- La risposta è che il metodo **`hashCode()`** è l'unica strada conosciuta per ottenere facilmente implementazioni efficienti. E' una scelta chiaramente:
  - a favore del programmatore-*realizzatore*
    - ✓ e quindi dell'efficienza
  - a scapito del programmatore-*utilizzatore*
    - ✓ e quindi della semplicità di utilizzo

Due Metodi al Posto di Uno (2)

- Per ottenere implementazioni efficienti i calcoli effettuati dal metodo **`hashCode()`** dovrebbero essere meno onerosi rispetto a quelli effettuati da **`equals()`**
  - **`int hashCode()`** viene invocato sempre
  - **`boolean equals(Object o)`** solo in caso di collisioni

- Attenzione: un metodo **`hashCode()`** così definito:

```
@Override
public int hashCode() { return 0; }
```

  - ➢ sarebbe sempre corretto e per nulla oneroso
  - ✓ produrrebbe sempre collisioni! La tavola hash degenera in una lista concatenata di oggetti in perenne conflitto tra loro

Due Metodi al Posto di Uno (3)

- E se per errore non (ri)definiamo il metodo **`hashCode()`** pur avendo ridefinito il metodo **`equals()`**?
- Si eredita quello definito nella radice della gerarchia di classi Java, ovvero dalla classe **`java.lang.Object`**: finiamo per usare come codice hash l'indirizzo di memoria dell'oggetto (ogni oggetto posside un codice diverso)
  - Quindi oggetti distinti che un'applicazione reputa equivalenti, siccome possiedono codici hash diversi, finirebbero per essere considerati distinti anche se il metodo **`equals()`** afferma invece il contrario
  - Un semplice test aiuta ad evidenziare la pericolosità di questo tipo di errori difficilmente rilevabili...

Gestione Duplicati in `HashSet<E>`: Esempio

- Consideriamo la seguente classe
  N.B.: manca il metodo **`hashCode()`**

```
public class Persona {

   private String nome;

   Persona(String nome)      { this.nome = nome; }

   public String toString()  { return this.nome; }
   public String getNome()   { return this.nome; }

   @Override
   public boolean equals(Object o) {
       Persona that = (Persona)o;
        return this.getNome().equals(that.getNome());
   }

}
```

Test Gestione Duplicati in `HashSet<E>`

```
public class HashSetTest {
   @Test public void testAddDuplicatiEnon() {
       Set<Persona> s = new HashSet<>();
       assertEquals(0, s.size());
       Persona paolo = new Persona("Paolo");
       Persona valter = new Persona("Valter");
       assertTrue(s.add(paolo));
       assertEquals(1, s.size());
       assertTrue(c.add(valter));
       assertEquals(2, s.size());
       assertFalse(s.add(paolo));
       assertEquals(2, s.size());
       Persona paolo2 = new Persona("Paolo");
       assertFalse(s.add(paolo2));
       assertEquals(2, s.size());
    }
}
```

`equals()` Orfano di `hashCode()`

- Se eseguiamo il test "sorprendentemente" **fallisce**, perché alcuni duplicati non vengono rilevati anche se il metodo **`equals()`** stabilisce correttamente il criterio di equivalenza desiderato
- Il problema è dovuto al fatto che non abbiamo ridefinito coerentemente il metodo **`hashCode()`**
- proviamo a stampare i codici hash, inserendo nel codice:

```
Persona p1 = new Persona("Paolo");
Persona p2 = new Persona("Paolo");
System.out.println("hash code p1: " +p1.hashCode());
System.out.println("hash code p2: " +p2.hashCode());
```

Coerenza di `hashCode()` ed `equals()` (1)

- L'implementazione usata dalla macchina evidentemente assegna due codici hash diversi ai due oggetti **`Persona`** uguali (secondo la nostra definizione di **`equals()`**)
- Per ovviare al problema dobbiamo ridefinire nella classe **`Persona`** il metodo **`hashCode()`** coerente con il criterio di equivalenza stabilito dal metodo **`equals()`**
- Ogni qualvolta si scrive un metodo **`equals()`** è necessario (OBBLIGATORIO) scrivere anche il corrispondente metodo **`hashCode()`**

Coerenza di `hashCode()` ed `equals()` (2)

- In sintesi, per implementare efficacemente il metodo **`hashCode()`** nella produzione del codice hash conviene usare le *stesse* informazioni usate dal metodo **`equals()`**
- Esempio: se **`equals()`** si basa su nome e cognome, è bene che anche **`hashCode()`** utilizzi le stesse informazioni
  - ritornando ad esempio alla classe **`Persona`**, si usa **`hashCode()`** del nome
  - avessimo nome e cognome, con **`equals()`** definito su questi due, un semplice metodo **`hashCode()`** potrebbe essere la somma degli **`hashCode()`** delle due stringhe

Gestione Duplicati in `HashSet<E>`: Esempio

- Aggiungiamo un opportuno metodo **`hashCode()`**

```
public class Persona {
   private String nome;

   Persona(String nome)      { this.nome = nome; }
   public String toString()  { return this.nome; }
   public String getNome()   { return this.nome; }

   @Override
   public int hashCode() {
          return this.getNome().hashCode();
   }

   @Override
   public boolean equals(Object o) {
       Persona that = (Persona)o;
       return this.getNome().equals(that.getNome());
   }
}
```

Test Gestione Duplicati in `HashSet<E>`

- Se facciamo rigirare **`HashSetTest`**, con la nuova definizione della classe **`Persona`**, possiamo osservare che ora (correttamente) non vengono inseriti duplicati nella collezione
- A confermare la natura del problema visto in precedenza, ristampiamo anche i codici hash così come forniti dal nuovo metodo **`hashCode()`**
  - ✓ ora oggetti distinti ma che consideriamo equivalenti possiedono lo stesso codice hash

`hashCode()`

- Tutte le classi della libreria standard hanno una implementazione del metodo **`hashCode()`**
- Per scrivere il metodo **`hashCode()`** delle nostre classi conviene usare una combinazione (ad esempio la somma) dei codici hash restituiti dai metodi **`hashCode()`** delle variabili di istanza già utilizzate nell'implementazione di **`equals()`**
  - nell'esempio precedente abbiamo usato il valore restituito dal metodo **`hashCode()`** del **`nome`**, che è di tipo **`String`**

Insiemi: Implementazione `HashSet<E>`

- Ricapitoliamo:
  - Se usiamo una collezione **`HashSet<E>`**:
    - gli elementi della collezione devono avere i metodi **`hashCode()`** e **`equals()`** appropriatamente definiti e in maniera mutuamente compatibile affinché **`hashCode()`** realizzi una funzione di hash rispetto al criterio di equivalenza stabilito da **`equals()`**
  - Indipendentemente dal fatto che gli oggetti della classe saranno elementi di un **`HashSet`**:
    - ogni qualvolta si scrive un metodo **`equals()`** è necessario (leggi OBBLIGATORIO) scrivere anche il corrispondente metodo **`hashCode()`** e viceversa

Sommario

- L'interface **`Set<E>`**
- Criteri equivalenza tra elementi
- **`HashSet<E>`**
  - Tavole Hash (intuizione)
  - **`hashCode()`**, **`equals()`**
- **`TreeSet<E>`**
  - Alberi binari di ricerca
  - **`compare()`**, **`compareTo()`**

Insiemi: Criterio di Equivalenza

- Due implementazioni di **`Set<E>`** e di **`SortedSet<E>`**, rispett., nel JCF:
  - **`HashSet<E>`**
  - **`TreeSet<E>`**
- Nelle due classi concrete il criterio di equivalenza tra elementi non è definito nella stessa maniera:
  - **`HashSet<E>`**
    si basa sui metodi **`equals()`** e **`hashCode()`**
  - **`TreeSet<E>`**
    si basa sul metodo **`Comparable.compareTo()`** o **`Comparator.compare()`**

`Set<E>`: Implementazione `TreeSet<E>` (1)

- L'implementazione **`TreeSet<E>`** garantisce (oltre all'assenza di duplicati) che gli elementi siano ordinati in accordo ad un criterio di ordinamento
  - l'ordinamento naturale degli elementi, *oppure*
  - un ordinamento stabilito da un comparatore esterno, ma noto all'insieme stesso
    - ovvero ottenuto al momento della creazione dell'insieme attraverso uno dei costruttori

Set<E>: Implementazione TreeSet<E> (2)

- Costruttori:
  - **`TreeSet()`**: Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
  - **`TreeSet(Collection<? extends E> c)`**: Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.
  - **`TreeSet(Comparator<? super E> comparator)`**: Constructs a new, empty tree set, sorted according to the specified comparator.

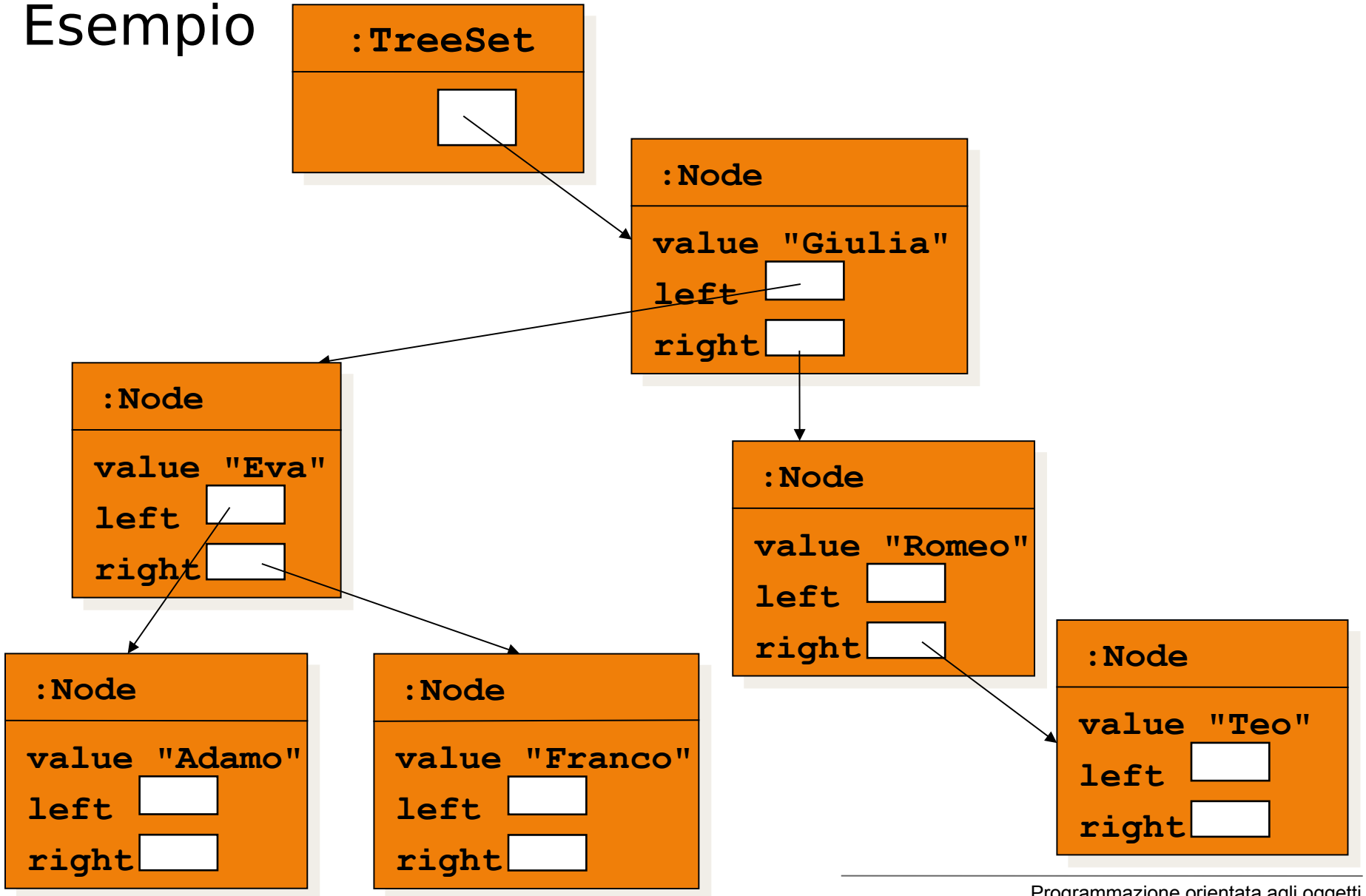
✓ Attenzione: se gli elementi non implementano **`Comparable<E>`** (o se non viene usato un comparatore **`Comparator<E>`** in fase di costruzione dell'oggetto **`TreeSet<E>`**) si può sollevare un errore a tempo di esecuzione anche se il codice compila!

`TreeSet<E>`: Alberi di Ricerca Binari (1)

- L'implementazione **`TreeSet`** è basata su alberi di ricerca binari
  - gli elementi dell'albero sono nodi che possiedono riferimenti ad ai nodi *figli*
  - in un albero binario ogni nodo ha due figli, *nodo destro* e *nodo sinistro,* che rispettano la seguente proprietà:
    - i valori di tutti i discendenti alla sinistra di qualunque nodo sono inferiori al valore del dato memorizzato in quel nodo, mentre tutti i discendenti alla destra contengono valori maggiori

TreeSet<E>: Alberi di Ricerca Binari (2)

- Esempio



`TreeSet<E>`: Alberi di Ricerca Binari (3)

- Per utilizzare una simile struttura dati è necessario che gli elementi si possano confrontare ed ordinare
  - per stabilire se un nodo deve andare a destra o a sinistra dobbiamo sapere se i suoi dati sono minori o maggiori dei dati del nodo genitore
- A tal fine, gli elementi devono implementare l'interface **`Comparable<E>`**
  - in alternativa è possibile che **`TreeSet<E>`** affidi la gestione di questo aspetto ad un comparatore (implementazione di **`Comparator<E>`**) esterno

Gestione Duplicati in `TreeSet<E>`: Esempio (1)

- Consideriamo nuovamente la classe **`Persona`**:

```
public class Persona {
   private String nome;

   Persona(String nome)      { this.nome = nome; }
   public String toString()  { return this.nome; }
   public String getNome()   { return this.nome; }

   @Override
   public int hashCode() {
       return this.getNome().hashCode();
   }

   @Override
   public boolean equals(Object p) {
       return this.getNome().equals(((Persona)p).getNome());
   }
}
```

Test Gestione Duplicati in TreeSet<E>

```
public class TreeSetTest {
   @Test public void testAddDuplicatiEnonConClassCastExc() {
      Set<Persona> s = new TreeSet<Persona>();

      Persona paolo = new Persona("Paolo");
      Persona valter = new Persona("Valter");
      assertTrue(s.add(paolo));
      assertTrue(s.add(valter));
      assertFalse(s.add(valter));
       assertEquals(2, s.size());
       Iterator<Persona> it = s.iterator();
       assertSame(paolo,  it.next());
       assertSame(valter, it.next());
   }
}
```

ClassCastException
Non funziona!

`TreeSet<E>`: Criterio di Equivalenza non Definito

- Se compiliamo e facciamo girare il codice precedente, si verifica un errore a tempo di esecuzione
- L'errore per un programmatore inesperto può essere di non facile interpretazione: è una **`ClassCastException`**
- Il problema nasce dal fatto che un criterio di ordinamento NON è stato specificato:
  - Né **`Persona`** implementa **`Comparable`**
  - Né il **`TreeSet`** riceve un comparatore usando un costruttore che permetta di specificarlo al momento della sua creazione

Gestione Duplicati in `TreeSet<E>`: Esempio (2)

- La classe ora viene modificata per implementare **`Comparable<Persona>`**

```
public class Persona implements Comparable<Persona> {
   private String nome;
   Persona(String nome)      { this.nome = nome; }
   public String toString() { return this.nome; }
   public String getNome()  { return this.nome; }
   @Override
   public int hashCode() {
           return this.getNome().hashCode();
   }
   @Override
   public int compareTo(Persona that) {
           return this.getNome().compareTo(that.getNome());
   }
   @Override
   public boolean equals(Object o) {
           return this.getNome().equals(((Persona)o).getNome());
   }
```

Gestione Duplicati in `TreeSet<E>`: Esempio (3)

- In alternativa: la classe **`Persona`** non viene fatta implementare **`Comparable<Persona>`**, ma specifichiamo il criterio di ordinamento passando un oggetto **`Comparator<Persona>`** al costruttore dell'insieme

```
public class Persona {  // N.B. non implementa Comparable<Persona>
   private String nome;

   Persona(String nome)      { this.nome = nome; }
   public String toString()  { return this.nome; }
   public String getNome()   { return this.nome; }

}

public class ComparatorePersone implements Comparator<Persona> {

   @Override
   public int compare(Persona p1, Persona p2) {
           return p1.getNome().compareTo(p2.getNome());
   }
}
```

Gestione Duplicati in TreeSet<E>: Esempio (4)

```
public class TreeSetTest {
   @Test public void testAddDuplicatiEnonConClassCastExc() {
      ComparatorePersone cmp = new ComparatorePersone();
      Set<Persona> s = new TreeSet<Persona>(cmp);

      Persona paolo = new Persona("Paolo");
      Persona valter = new Persona("Valter");
      assertTrue(s.add(paolo));
      assertTrue(s.add(valter));
      assertFalse(s.add(valter));
       assertEquals(2, s.size());
       Iterator<Persona> it = s.iterator();
       assertSame(paolo,  it.next());
       assertSame(valter, it.next());
   }
}
```

Gestione Duplicati in `TreeSet<E>`: Esempio (5)

- ✓ per implementare **`ComparatorePersone`** ci siamo basati sul fatto che **`java.lang.String`** a sua volta implementa **`Comparable<String>`**
- A questo punto se facciamo girare nuovamente il codice di **`TreeSetTest`**, il comportamento è quello atteso:
  - viene fatto un solo inserimento
  - non si verificano errori a tempo di esecuzione,
  - gli elementi sono ordinati secondo l'ordinamento naturale (quello lessicografico delle stringhe)

`TreeSet<E>`: Gestione Duplicati

- Riassumendo, per verificare la presenza di duplicati (e per mantenere ordinata la collezione)
  - **`TreeSet<E>`** usa il metodo **`compareTo()`** quindi gli oggetti che appartengono alla collezione devono implementare **`Comparable<E>`**
  - Oppure l'oggetto **`TreeSet<E>`** deve essere creato passando al costruttore un oggetto **`Comparator<E>`**

Eguaglianza e `Set<E>`

- Ovviamente le implementazioni di **`compareTo()`**, **`hashCode()`** e **`equals()`** devono avere una semantica coerente
  - Ogni volta che definiamo il metodo **`equals()`** dobbiamo definire anche il metodo **`hashCode()`**
  - Se definiamo il metodo **`compareTo()`** (cioè se la classe implementa **`Comparable<E>`**), questo è bene che sia coerente con **`equals()`**
- Dati due riferimenti ad oggetti **`x`** e **`y`**, allora

`x.compareTo(y)` restituisce `0`

se e solo se

`x.equals(y)` restituisce **`true`**

Implementazioni di `Set<E>`: Verifica Duplicati

- Riassumendo:
  - **`TreeSet<E>`** verifica l'esistenza di duplicati
    - tramite il metodo **`compareTo()`**, ed il tipo **`E`** deve implementare l'interface **`Comparable<E>`**;
    - *oppure*, tramite il metodo **`compare()`** dell'interfaccia **`Comparator`** una cui istanza gli è stata fornita all'atto dell'istanziamento (vedi i costruttori di **`TreeSet<E>`** nella documentazione)
  - **`HashSet<E>`** verifica l'esistenza di duplicati tramite i metodi **`hashCode()`** e **`equals()`** del tipo **`E`**

`Set<E>`: Quale Implementazione?

- **`TreeSet<E>`** mantiene ordinata la collezione, ma sia gli aggiornamenti che gli accessi casuali risultano meno efficienti della controparte non ordinata
  - ✓ perché bisogna mantere l'ordinamento della rappresentazione sottostante
  - – va utilizzata solo se esiste l'esigenza vera di mantenere ordinata la collezione
- Altrimenti preferire la più efficiente **`HashSet<E>`**
- Questi aspetti sono stati affrontati approfonditamente nel corso di *Algoritmi e Strutture Dati*

Esercizio

- Date le classi **`Studente`** e **`Aula`**

```
public class Studente {
    private String nome;

    public Studente(String nome) { this.nome = nome; }
    public String getNome() { return this.nome; }
}

public class Aula {
    private Set<Studente> studenti;
    public Aula() {  /* scrivere il codice */  }
    public boolean addStudente(Studente studente) {
      // scrivere il codice
    }
}
```

Esercizio (cont.)

- Scrivere il codice del costruttore di **`Aula`** e del metodo **`addStudente(Studente s)`**
- Domanda 1:
  - produrre una soluzione considerando che è possibile modificare la classe **`Studente`**
- Domanda 2:
  - produrre una soluzione considerando che NON è possibile modificare la classe **`Studente`**, ma è possibile definire altre classi
- Domanda 3:
  - come la domanda 1, ma l'insieme degli studenti deve essere ordinato e non è possibile introdurre altre classi

Una Soluzione alla Domanda 1

```
public class Studente {
    private String nome;
    public Studente(String nome) { this.nome = nome; }
    public String getNome() {return this.nome; }
    @Override
    public int hashCode() { return this.nome.hashCode(); }
    @Override
    public boolean equals(Object o) {
       Studente that = (Studente)o;
       return this.getNome().equals(that.getNome());
    }
}
public class Aula {
    private Set<Studente> studenti;
    public Aula(){ this.studenti = new HashSet<Studente>(); }
    public boolean addStudente(Studente studente) {
       return this.studenti.add(studente);
    }
```

Una Soluzione alla Domanda 2

```
public class Studente {
   private String nome;

   public Studente(String nome) {this.nome = nome;}
   public String getNome() {return this.nome;}
}

public class Aula {
   private Set<Studente> studenti;
   public Aula() {
       this.studenti = new TreeSet<>(new ComparatoreStudenti());
   }
   public boolean addStudente(Studente studente) {
       return this.studenti.add(studente);
   }
}
```

Una Soluzione alla Domanda 2 (cont.)

```
import java.util.*;

public class ComparatoreStudenti
             implements Comparator<Studente> {
   @Override
   public int compare(Studente s1, Studente s2) {
       return s1.getNome().compareTo(s2.getNome());
   }
}
```

Una Soluzione alla Domanda 3

```
public class Studente implements Comparable<Studente> {
    private String nome;
    public Studente(String nome) { this.nome = nome; }
    public String getNome()      { return this.nome; }
    @Override
    public int hashCode() { return this.getNome().hashCode(); }
    @Override
    public boolean equals(Object o) {
       Studente that = (Studente)o;
       return this.getNome().equals(that.getNome());
    }
    @Override
    public int compareTo(Studente that) {
       return this.getNome().compareTo(that.getNome());
    }
}
public class Aula {
    private Set<Studente> studenti;
    public Aula() { this.studenti = new TreeSet<Studente>(); }
    public boolean addStudente(Studente studente) {
       return this.studenti.add(studente);
    }
}
```

Suggerimento

- Eclipse genera automaticamente il codice dei metodi **`equals()`** e **`hashCode()`**
  - ✓ si osservi che vengono generati entrambi contestualmente
  - ✓ è sufficiente indicare su quali variabili di istanza deve basarsi il criterio di equivalenza

- Durante l'apprendimento NON UTILIZZARLO!

✓ Meglio mantenere la piena consapevolezza del codice utilizzato e, *soprattutto*, di come scriverlo

`NavigableSet<E>`: da Java 6

- In Java 6 le funzionalità offerte da **`SortedSet<E>`** sono state rinconsiderate ed estese
- Allo scopo si aggiunta una specializzazione di **`SortedSet<E>`** denominata **`NavigableSet<E>`**
  - ✓ implementata anche da **`TreeSet<E>`**
- Qualche semplice esempio di metodi di cui si avvertiva la necessità (consultare i javadoc):

`E pollFirst() / E pollLast()`

Retrieves and removes the first (lowest) [last (highest)] element, or returns null if this set is empty

`NavigableSet<E> descendingSet()`

Returns a reverse order view (>>) of the elements contained in this set

`NavigableSet<E>`: Viste *Attive*

Un esempio più articolato:

```
NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                       E toElement, boolean toInclusive)
```

Returns a view of the portion of this set whose elements range from **`fromElement`** *to* **`toElement`**.

If **`fromElement`** *and* **`toElement`** *are equal, the returned set is empty unless* **`fromInclusive`** *and* **`toInclusive`** *are both true*

- Questo metodo, ed altri strettamente correlati (cfr. **`headSet()`**, **`tailSet()`**), producono come risultato delle «viste attive» sull'insieme ordinato di partenza
  - *The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa.*

✔ Cambiamenti della vista si riflettono sull'insieme originale

Se non è il comportamento desiderato, basta «materializzare» la vista usandola come argomento del costruttore di una nuova collezione

NavigableSet<E> e Retrocompatibilità

- ✓ Alcune scelte sembrano discutibili, ma vanno certamente valutate nel contesto in cui sono nate, ovvero con l'esigenza di mantenere la retro-compatibilità con le implementazioni e le interfacce già esistenti e diffuse

```
@Override
SortedSet<E> headSet(E toElement)
```

- *Returns a view of the portion of this set whose elements are strictly less than* ***`toElement`***.

```
NavigableSet<E>
                headSet(E toElement, boolean inclusive)
```

- *Methods* ***`subSet(E, E)`***, ***`headSet(E)`***, *and* ***`tailSet(E)`*** *are specified to return* ***`SortedSet`*** *to allow existing implementations of* ***`SortedSet`*** *to be compatibly retrofitted to implement* ***`NavigableSet`***, *but extensions and implementations of this interface are encouraged to override these methods to return* ***`NavigableSet`***