

# Programmazione Orientata agli Oggetti

---

Collezioni:  
Insiemi generici

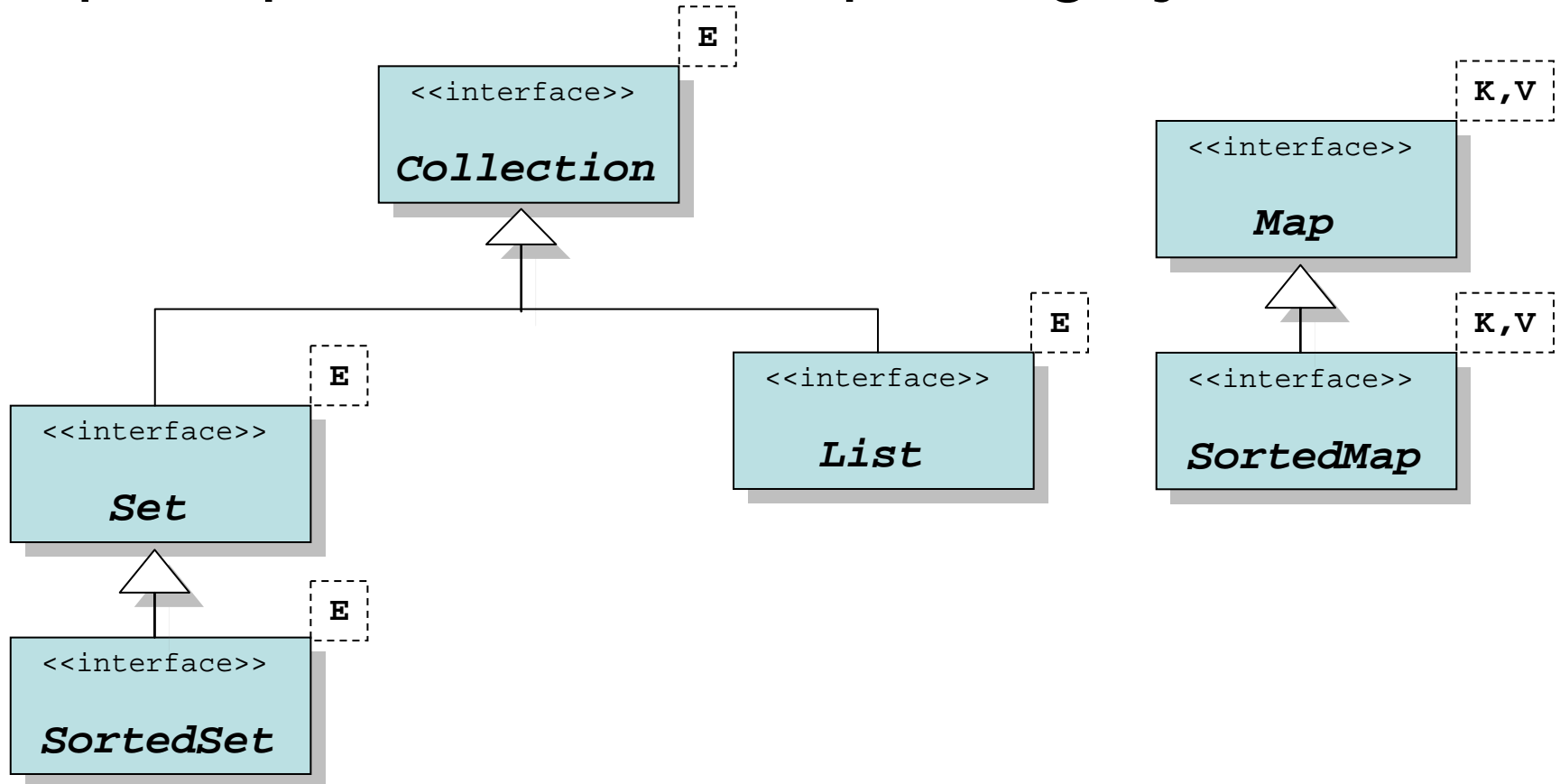
# Concetti introdotti

---

- Insiemi, criteri di equivalenza tra elementi
- Tavole Hash
- **Set<E>**, metodi **hashCode()**, **equals()**
- **TreeSet<E>**, metodi **compare()**, **compareTo()**

# Collezioni: 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 può contenere *duplicati*
- L'interface `Set<E>`, che estende l'interface `Collection<E>`, offre tutti e soli i metodi della interface `Collection<E>`, con la restrizione che le classi che la implementano si impegnano a non ammettere la presenza di elementi *duplicati*
- E' necessario stabilire un criterio di equivalenza tra elementi ai fini dell'insieme

# Insiemi: implementazioni

---

- Nelle librerie del Collection Framework abbiamo due implementazioni di `Set<E>`:
  - `HashSet<E>`
  - `TreeSet<E>`
- Per dettagli puramente implementativi, 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 `compareTo()` o `compare()` (di una classe esterna)

# Criterio di equivalenza

---

- 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 un criterio di equivalenza tra oggetti
- `HashSet<E>` richiede che tali classi definiscano opportunamente due metodi che servono a verificare ed evitare la presenza di duplicati
  - il metodo *equals*  
`public boolean equals(Object that)`
  - il metodo *hashCode*  
`public int hashCode()`

# Insiemi: implementazione `HashSet<E>`

---

- Per comprendere le motivazioni alla base di questa scelta è necessario discutere alcuni dettagli relativi all'implementazione
- L'implementazione di `HashSet<E>` si basa su tavole hash
- Vediamo intuitivamente di cosa si tratta

# Insiemi: implementazione con tavole hash

---

- Una *funzione di hash* è una funzione che calcola un numero intero, *codice hash*, a partire dai dati di un oggetto, in modo che sia molto probabile che oggetti *non eguali* abbiano codici diversi
- Due o più oggetti possono avere lo stesso codice di hash: questa situazione genera una *collisione*
- Una buona funzione di hash deve minimizzare le collisioni



# Codici hash

---

- Esempi di codici hash per una stringa
  - Il numero di caratteri che compongono la stringa
    - "Pippo": 5
    - "Pluto": 5
    - "Paperino": 8
    - Non e' un buon codice: 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":  $80+65+80+69+82+73+78+79=626$
    - Va meglio ...
- Osservazione fondamentale: se i codici hash sono diversi allora le stringhe non sono eguali; il contrario non è necessariamente vero

# Tavole hash

---

- Le funzioni di hash vengono usate per creare collezioni (mappe e insiemi) molto efficienti
- L'idea è quella di avere un array, i cui indici siano i codici hash degli elementi

...	
392	PLUTO
...	
...	
400	PIPPO
...	
626	PAPERINO
...	

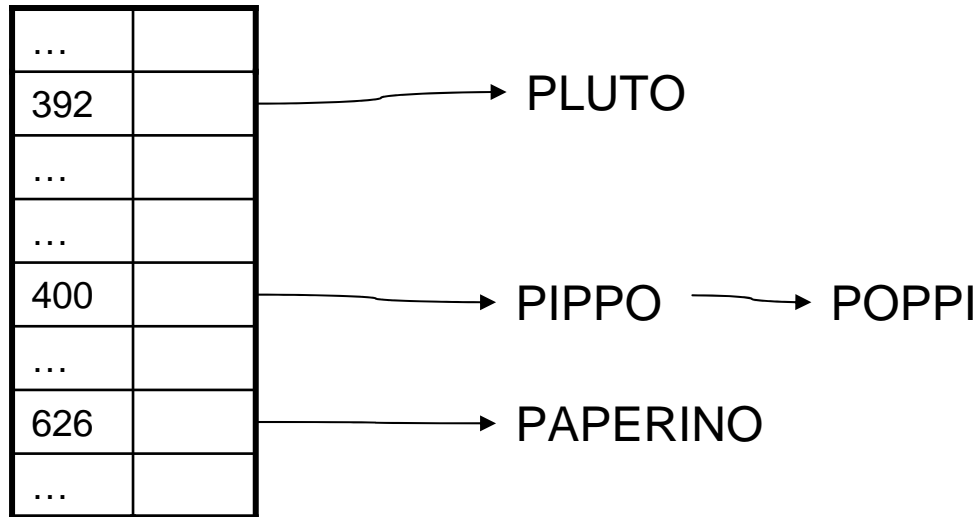
# Tavole hash

---

- Questa idea però ha due problemi
  - La dimensione dell'array
  - Le collisioni
- Per ovviare al primo problema si deve scegliere una funzione di hash che generi codici in un range ragionevolmente ridotto
- Per ovviare al secondo problema si usano liste concatenate

# Tavole hash

---



- Se la funzione di hash è ben definita (range piccolo e poche collisioni) le operazioni di verifica di appartenenza, rimozione e inserimento in una collezione hanno costo costante

# Tavole hash

---

- Le implementazioni delle mappe e degli insiemi usano tavole hash molto sofisticate
- In Java il codice hash di un oggetto deve essere restituito dal metodo  
`int hashCode()`
- Se vogliamo usare l'implementazione `HashSet<E>` di `Set<E>`, gli oggetti della collezione devono avere i metodi `hashCode()` e metodo `equals()` definiti opportunamente
  - Se non li definiamo, questi metodi vengono ereditati da `java.lang.Object` secondo una semantica che, di solito, è inutile se non dannosa: `hashCode()` restituisce l'indirizzo in memoria, `equals()` equivale a `==`

# Eguaglianza: il metodo `equals()`

---

- Per ogni classe andrebbe definito accuratamente il metodo `equals()`
- Il metodo `equals()` deve soddisfare le proprietà:
  - Riflessività: per qualsiasi riferimento `x`, `x.equals(x)` deve restituire `true`
  - Simmetria: per qualsiasi riferimento `x` e `y`, `x.equals(y)` deve restituire `true` se e solo se `y.equals(x)` restituisce `true`
  - Transitività: per qualsiasi riferimento `x`, `y` e `z`, se `x.equals(y)` restituisce `true` e `y.equals(z)` restituisce `true`, allora `x.equals(z)` deve restituire `true`
  - Per qualsiasi riferimento `x` non nullo, `x.equals(null)` restituisce `false`
  - Se due riferimenti `x` e `y` sono identici (`x==y`), allora `x.equals(y)` deve restituire `true`

# Gestione duplicati in `HashSet<E>`

---

- In sintesi, per verificare la presenza di duplicati `HashSet<E>` usa `equals()` in combinazione con il valore restituito dal metodo `hashCode()`.
- In particolare, per verificare se due oggetti sono eguali, `HashSet` prima verifica se il loro codice hash è identico: solo in caso affermativo (collisione) viene invocato il metodo `equals()`
  - per questo motivo è meglio resistere alla tentazione di scrivere parte di `equals()` in funzione di `hashCode()`: sarebbe tutto lavoro sprecato
  - conviene che il metodo `hashCode()` risulti meno oneroso rispetto ad `equals()`

# Due metodi al posto di uno

---

- Attenzione che un metodo `hashCode()` così definito sarebbe sempre corretto:  

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

  - con questa definizione ci sono *sempre* collisioni
- Ma allora 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 per ottenere implementazioni efficienti
  - è una scelta a favore del programmatore realizzatore (e quindi dell'efficienza) ed a scapito del programmatore utilizzatore (e quindi della semplicità di utilizzo)



# Gestione duplicati in HashSet<E>: esempio

---

- Consideriamo la seguente classe

```
public class Persona {  
  
    private String nome;  
  
    Persona(String nome)    { this.nome = nome; }  
    public String toString() { return this.nome; }  
    public String getNome() { return this.nome; }  
  
    public boolean equals(Object p) {  
        return this.nome.equals(((Persona)p).getNome());  
    }  
}
```

# Gestione duplicati in HashSet<E>: test

---

```
public class HashSetTest {  
    @Test public void testAddDuplicatiEnon() {  
        Set<Persona> c = new HashSet<Persona>();  
        assertEquals(0, c.size());  
        Persona paolo = new Persona("Paolo");  
        Persona valter = new Persona("Valter");  
        assertTrue("Aggiunto Paolo", c.add(paolo));  
        assertEquals(1, c.size());  
        assertTrue("Aggiunto Valter", c.add(valter));  
        assertEquals(2, c.size());  
        assertFalse("dup. stesso oggetto", c.add(paolo));  
        assertEquals(2, c.size());  
        Persona paolo2 = new Persona("Paolo");  
        assertFalse("dup. nuovo oggetto", c.add(paolo2));  
        assertEquals(2, c.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 definito il metodo `hashCode()`
- provare 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());
```

# hashCode ( ) ed equals ( ) compatibili

---

- L'implementazione usata dalla macchina evidentemente assegna due codici hash diversi ai due oggetti persona eguali
- Per ovviare al problema dobbiamo definire nella classe **Persona** un corpo del metodo **hashCode ( )** compatibile con il criterio di equivalenza stabilito dal metodo **equals ( )**
- Ogni qualvolta si scrive un metodo **equals ( )**, è necessario, immancabilmente, scrivere anche il corrispondente metodo **hashCode ( )**

# Gestione duplicati in HashSet<E>: esempio

---

- Aggiungiamo 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; }  
  
    public int hashCode() {  
        return this.nome.hashCode();  
    }  
  
    public boolean equals(Object p) {  
        return this.nome.equals(((Persona)p).getNome());  
    }  
}
```

# Gestione duplicati in `HashSet<E>`: esempio

---

- Se facciamo girare nuovamente il codice di test in `HashSetTest`, con la nuova definizione della classe `Persona`, possiamo osservare che correttamente non vengono inseriti duplicati nella collezione
- Per prova, facciamo nuovamente stampare anche il codice hash così come implementato dal nostro metodo ed osserveremo che ora oggetti eguali ma distinti possiedono stesso codice hash

# hashCode ( )

---

- Tutte le classi della libreria standard hanno una implementazione (ben definita) 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
  - Nell'esempio precedente abbiamo usato il valore restituito dal metodo `hashCode ( )` della variabile `nome`, che è di tipo `String`

# Insiemi: implementazione `HashSet<E>`

---

- Se usiamo una collezione `HashSet<E>` allora:
  - 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()`
- Di nuovo: ogni qualvolta si scrive un metodo `equals()`, è necessario, immancabilmente, scrivere anche il corrispondente metodo `hashCode()` e viceversa



# Set<E>: implementazione TreeSet<E>

---

- L'implementazione **TreeSet<E>** garantisce (oltre all'assenza di duplicati) che gli elementi siano ordinati in accordo a
  - l'ordinamento naturale interno, oppure
  - un ordinamento esterno stabilito da un comparatore e noto all'insieme stesso (ad es. perché ricevuto al momento della sua creazione attraverso uno dei suoi molteplici costruttori)
- Vedi documentazione
- 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!

# Gestione duplicati in TreeSet<E>: esempio

---

- Consideriamo nuovamente la classe

```
public class Persona {  
    private String nome;  
  
    Persona(String nome)    { this.nome = nome; }  
    public String toString() { return this.nome; }  
    public String getNome() { return this.nome; }  
  
    public int hashCode() {  
        return this.nome.hashCode();  
    }  
  
    public boolean equals(Object p) {  
        return this.nome.equals(((Persona)p).getNome());  
    }  
}
```

# Gestione duplicati in TreeSet<E>: test

---

```
public class TreeSetTest {  
    @Test public void testAddDuplicatiEnonConClassCastExc() {  
        Set<Persona> c = new TreeSet<Persona>();  
        Persona paolo = new Persona("Paolo");  
        Persona valter = new Persona("Valter");  
        assertTrue("Aggiunto Paolo",c.add(paolo));  
        assertTrue("Aggiunto Valter",c.add(valter));  
        assertFalse("Aggiunto duplicato",c.add(valter));  
        assertEquals("Ci sono dispersi?", 2, c.size());  
        Iterator<Persona> it = c.iterator();  
        assertSame("e Paolo?", paolo, it.next());  
        assertSame("e Valter?", valter, it.next());  
    }  
}
```

## **TreeSet<E>: criterio di equivalenza non definito**

---

- Se compiliamo e facciamo eseguire 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 non si verifica nessuna delle due condizioni (una delle due è sufficiente ad evitare l'errore):
  - **Persona** non implementa **Comparable**
  - **TreeSet** *c* all'atto della creazione non riceve nessun comparatore nel costruttore

# Gestione duplicati in TreeSet<E>: esempio

---

- La classe implementa 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; }

    public int hashCode() {
        return this.nome.hashCode();
    }

    public int compareTo(Persona p) {
        return this.nome.compareTo(p.getNome());
    }

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

# Gestione duplicati in `TreeSet<E>`: esempio

---

- Ci siamo basati sul fatto che `java.lang.String` 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( )** e, 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()`, e la classe `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'istanziamiento
  - `HashSet<E>` verifica l'esistenza di duplicati tramite i metodi `hashCode()` e `equals()` della classe `E`

# Set<E>: quale implementazione?

---

- **TreeSet<E>** è meno efficiente, e pertanto va utilizzata solo se esiste l'esigenza di mantenere ordinata la collezione
- Altrimenti **HashSet<E>**