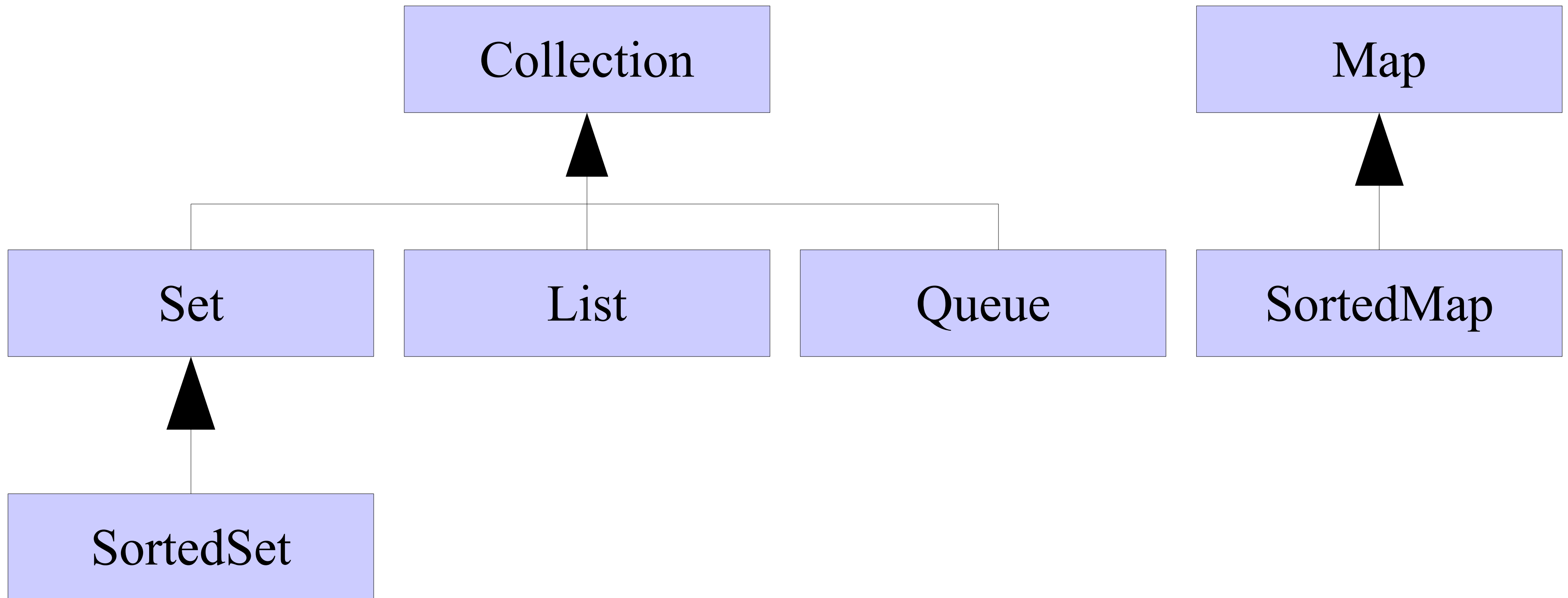
# Gerarchia di Collection



# Gerarchia di Collection

- Vediamo un po' più in dettaglio solo **`Set<E>`** e **`SortedSet<E>`**
- Per le altre interfacce si possono consultare le API o il tutorial sulle collections (sul sito **`http://java.sun.com`**)

# Set<E>

- Abbiamo visto `List<E>` per mantenere una collezione di oggetti con ripetizione
- L'interface `Set<E>` invece modella esattamente il **concetto matematico di insieme**
- Rappresenta una collezione di oggetti di tipo `E` in cui un oggetto è presente una sola volta oppure non è presente
- Se si aggiunge un oggetto già presente il `Set` non cambia

# Interfaccia di Set<E>

```
public interface Set<E> extends Collection<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();
    ...
```

# Appartenenza

- Il metodo **`contains`** è l'analogo di **`indexOf`** di **`List<E>`**
- Qui restituisce **`true`** se presente e **`false`** se non presente, dato che **in un insieme non c'è il concetto di posizione**
- Valgono le stesse considerazioni fatte riguardo alla ridefinizione del metodo **`equals`**: l'appartenenza viene stabilita scorrendo gli elementi dell'insieme e confrontandoli, tramite il metodo **`equals`**, con l'oggetto passato

# Inserimento e rimozione

- I metodi **`add`** e **`remove`** restituiscono un **`boolean`** che indica se l'operazione ha modificato il **`Set<E>`**
- Nel caso di **`add`** l'aggiunta di un oggetto già presente (secondo il metodo **`equals`** della classe **`E`**) non cambia il **`Set<E>`**
- Nel caso di **`remove`** l'eliminazione di un oggetto non presente (secondo il metodo **`equals`** della classe) non cambia il **`Set<E>`**
- L'iteratore è analogo a quello di **`List<E>`** tranne che qui l'ordine non ha un significato particolare

# Classe standard di implementazione

- La classe standard del pacchetto **`java.util`** che implementa l'interface **`Set<E>`** è la classe **`HashSet<E>`**
- Fa uso di tecniche di hash: è quindi assolutamente necessario ridefinire correttamente il metodo **`hashCode`** della classe **`E`** se si ridefinisce **`equals`**
- Una classe alternativa è **`TreeSet<E>`** che usa alberi particolari che mantengono un ordinamento

# SortedSet<E>

- L'interfaccia **`SortedSet<E>`** modella un insieme in cui **gli elementi hanno un ordinamento**
- Vale qui il concetto di ordinamento naturale espresso dall'interfaccia **`Comparable<E>`** che abbiamo visto anche in **`List<E>`**
- Però se si vuole utilizzare **`SortedSet<E>`** è assolutamente necessario che l'ordinamento naturale di **`E`** sia compatibile con **`equals`** di **`E`**

# Interface SortedSet<E>

```
public interface SortedSet<E> extends Set<E> {
    //Tutti i metodi di Set<E> più:
    SortedSet<E> subSet(E fromElement,
                                   E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    E first(); // Minimo
    E last();  // Massimo
```

# Classe implementazione

- La classe di implementazione standard per **`SortedSet<E>`** è **`TreeSet<E>`**
- L'iteratore di un **`SortedSet<E>`** scorre gli elementi dell'insieme **secondo l'ordinamento naturale** di **`E`**, dal più piccolo al più grande
- Un punto di forza dell'uso di questa interfaccia sta nel fatto che durante una ricerca possiamo **evitare di iterare su tutti gli elementi** fermandoci quando abbiamo oltrepassato il punto dove dovrebbe essere l'elemento cercato secondo l'ordinamento naturale

# Esempio: prenotazioni di aule

- Modelliamo un dominio del discorso in cui una comunità di utenti ha a disposizione un certo insieme di aule con certe caratteristiche
- Ogni utente può prenotare un'aula per un certo giorno, in una certa ora, per una certa attività
- Non ci devono essere sovrapposizioni nelle prenotazioni
- Definiamo una classe `GestoreAule` che mantiene un `Set<Aula>` contenente le aule

# GestoreAule

```
public class GestoreAule {
  private Set<Aula> aule;
  public GestoreAule() {
    aule = new HashSet<Aula>();
  }
  public boolean add(Aula a){
    if (a == null)
      return false;
    return aule.add(a);
  } ...
```

Usiamo l'implementazione standard

L'implementazione standard permette di inserire in un **`Set`** anche elementi **`null`**
In questo caso facciamo un controllo per evitarlo

# Prenotazione

- Una prenotazione viene fatta da un certo utente, in una certa data e ora per una certa attività

```
public class Prenotazione implements
  Comparable<Prenotazione> {

  // L'utente che effettua questa prenotazione

  private Utente utente;

  // Data e ora della prenotazione

  private DataEOra dataEOra;

  private String descrizione; ...
```

# Prenotazione

- La classe **`Prenotazione`** implementa l'interfaccia **`Comparable<Prenotazione>`**
- L'ordinamento naturale che definisce è quello cronologico
- Una prenotazione precede un'altra se è in una data e ora precedente
- Questo ci permette di usare un **`SortedSet<Prenotazioni>`** per mantenere le prenotazioni di **`Aula`** ordinate per data e ora

# Aula

- Ogni Aula ha il suo personale insieme di prenotazioni ordinato cronologicamente:

```
public class Aula {
  // Identificativo unico di un'aula
  private final String nome;
  ...
  // Insieme delle prenotazioni per
  quest'aula, // in ordine cronologico
  private SortedSet<Prenotazione> prenotazioni;
  ...
```

# Controllo Aula libera

- Per controllare se un'**`Aula`** è libera in una certa data e ora dovremmo esaminare tutte le prenotazioni dell'**`Aula`** e verificare che non ce n'è nessuna in quella data e in quella ora
- Usando l'ordinamento cronologico dell'insieme possiamo organizzare la ricerca in modo tale da evitare di guardare tutte le prenotazioni se non è necessario
- Basterà fermarsi dopo il punto in cui dovrebbe trovarsi la prenotazione, se ci fosse

# Controllo Aula libera

```
public boolean libera(DataEOra d) {
  Prenotazione p;
  DataEOra tmp;
  // Scorre l'insieme ordinato di prenotazioni
  e // vede se la data e l'ora sono occupate
  Iterator<Prenotazione> i =
                     prenotazioni.iterator();
  // Cerco una prenotazione nella stessa data
  e // nella stessa ora
  // Continua...
```

# Controllo Aula libera

```
boolean trovato = false;
boolean interrompi = false;
while (i.hasNext() && !trovato && !interrompi){
  p = i.next(); tmp = p.getDataEOra();
  if (tmp.equals(d))
     trovato = true;
  else if (tmp.compareTo(d) > 0)
       // Ho superato la data senza aver trovato
       interrompi = true;
    } // Continua...
```

# Controllo Aula libera

```
if (trovato)
        return false;
else return true;
}
```