UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
https://jdirocco.github.io

Sommario

- Introduzione alle collection
- Interfacce
- Implementazione
- Collection
- List
- Set
- Map
- Queue e Dequeue
- Ordinamento
  - Comparable
  - Comparator
- SortedSet e SortedMap
- Implementazioni Wrapper

Introduzione (1)

- Una collezione (chiamata anche *container*) è un oggetto che raggruppa elementi multipli in una singola unità
- Sono utilizzate per memorizzare, recuperare e manipolare dati, per trasmetterli da un metodo ad un altro
- Tipicamente rappresentano dati correlati tra loro, come una collezione di numeri telefonici, collezione di lettere, ecc
- Sono state introdotte a partire dalla release 1.2 (collection framework)
- In precedenza esistevano alcune classi utilizzate come container (`Vector, Hashtable`)

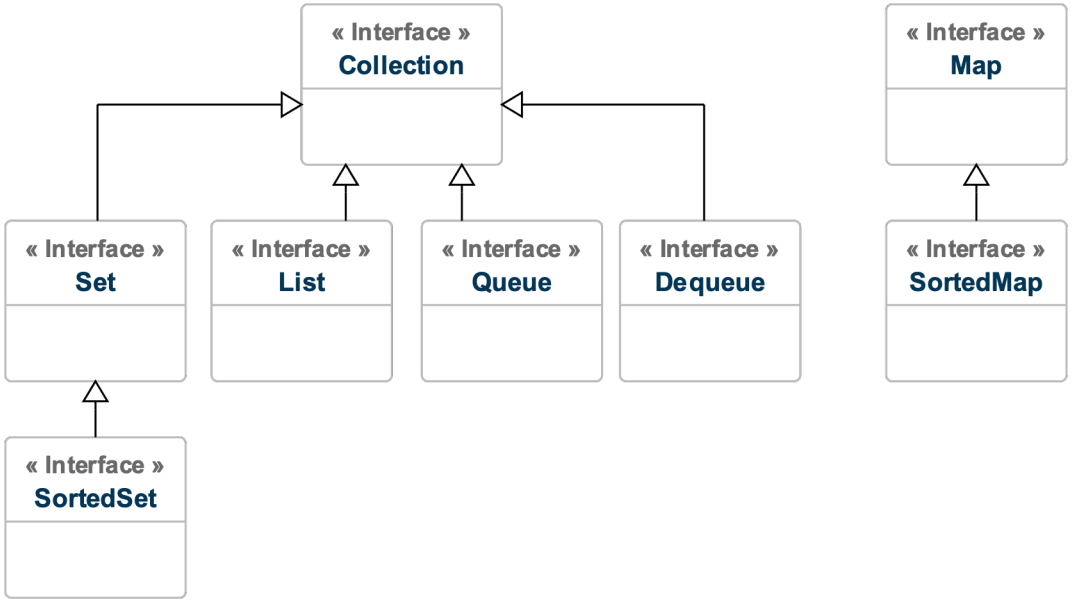
Introduzione (2)

- Un framework per le collezioni è composto in genere da
  - **Interfacce**
    - Tipi di dato astratti che rappresentano le collezioni
    - Permettono di manipolare le collezioni indipendentemente dai dettagli della rappresentazione
    - In genere formano una gerarchia
  - **Implementazioni**
    - Implementazioni concrete delle interfacce
    - Sono le strutture dati riusabili
  - **Algoritmi**
    - Metodi che effettuano delle computazioni sulle collezioni, come ad esempio ordinamento, ricerca, ...
    - Gli algoritmi sono polimorfici poiché gli stessi metodi possono essere applicati a differenti implementazioni
    - Sono le funzionalità riusabili

Introduzione (3)

- **Benefici**
  - Riduce lo sforzo di programmazione
  - Incrementa la velocità e qualità dello sviluppo
  - Permette l'interoperabilità tra API non in relazione
  - Riduce il tempo di apprendimento e l'utilizzo di nuove API
  - Riduce il tempo per lo sviluppo di nuove API
  - Aumenta il riuso di software
- **Svantaggi**
  - Generalmente sono abbastanza complesse
  - Collection di Java sono abbastanza semplici

Interfacce (1)



Nota

Non vengono fornite interfacce separate per ogni variante di ogni tipo di collection (es.: immutabili, dimensione fissa e solo in append)

Interfacce (2)

`Collection`

- Root della gerarchia
- Rappresenta un gruppo di oggetti conosciuti come elementi
- E' il minimo comun denominatore che tutte le collezioni implementano
- Alcune implementazioni
  - Ammettono duplicati altre no
  - Ordinamento su elementi oppure no
- JDK non ha implementazioni di tale interfaccia
- Non è possibile inserire valori di tipi primitivi
  - E' necessario utilizzare tipi wrapper `Integer`, `Long`
- Vengono fornite implementazioni delle sue sotto-interfacce come `Set`, `List`

Interfacce (3)

- `Set`
  - Collezione che **non può** contenere duplicati
  - Astrazione dell'*insieme* matematico
- `List`
  - Collezione ordinata (detta anche *sequenza*)
  - Può contenere elementi duplicati
  - Si accede agli elementi mediante un indice intero (*posizione*)
- `Queue`
  - Collezione utilizzata per mantenere elementi multipli in base ad un ordine (generalmente **FIFO**: First-in First-out)
  - Contiene operazioni aggiuntive di inserimento, estrazione ed ispezione

Interfacce (3)

- `Deque`
  - Collezione utilizzata per mantenere elementi multipli in base ad un ordine
  - Contiene operazioni aggiuntive di inserimento, estrazione ed ispezione
  - Può essere utilizzata sia come FIFO che LIFO (Last-in First-out)
- `Map`
  - Oggetto che mappa una chiave ad un valore
  - Non possono contenere chiavi duplicate ovvero una chiave mappa un solo valore

Interfacce (4)

- `SortedSet`
  - Insieme dove gli elementi sono ordinati in ordine ascendente
  - Operazioni aggiuntive per utilizzare l'ordinamento
- `SortedMap`
  - Map dove le chiavi sono ordinate in ordine ascendente

- L'ordine per `SortedSet` e le `SortedMap` viene stabilito all'atto dell'utilizzo di un'implementazione (`Comparator`) oppure dando l'ordine agli oggetti che contengono (`Comparable`)

Implementazione (1)

Interfacce	Implementazioni				
	Hash Table	**Resizable Array**	**Tree**	**Linked List**	**Hash table + Linked list**
Set	`HashSet`		`TreeSet`		`LinkedHashSet`
List		`ArrayList`		`LinkedList`	
Queue				`LinkedList`	
Deque (*)		`ArrayDeque`		`LinkedList`	
Map	`HashMap`		`TreeMap`		`LinkedHashMap`

(*) Altra implementazione `PriorityQueue`

Implementazione (2)

- Sono presenti almeno due implementazioni per ogni interfaccia
- Implementazioni primarie
  - `HashSet`
  - `ArrayList`
  - `HashMap`
- `TreeSet` e `TreeMap` implementano `SortedSet` e `SortedMap`
- `Vector` e `Hashtable` erano presenti prima dell'introduzione delle collection
  - Modificate per implementare le nuove interfacce

Implementazione (3)

- Ogni implementazione non è sincronizzata diversamente da `Hashtable` e `Vector`
  - Metodi wrapper che sincronizzano
- Sono permessi elementi, chiavi e valori `null`
- Tutte le implementazioni sono serializzabili (`java.io.Serializable`) e supportano il metodo `clone()`

Implementazione (4)

- `Set`
  - `HashSet` è molto più veloce di `TreeSet`
    - Tempo costante vs. tempo logaritmico
  - `HashSet` non garantisce l'ordinamento
  - `TreeSet` sì
  - `HashSet` necessità della capacità iniziale che ha impatto su performance
    - Default 101 che è sufficiente
    - Altrimenti costruttore appropriato
    - Vedere documentazione

Implementazione (5)

- `List`
  - `ArrayList` è più veloce di `LinkedList` poiché permette un accesso posizionale con tempo costante e non deve allocare un oggetto `Node` per ogni elemento nella Lista
  - Se vengono aggiunti frequentemente elementi all'inizio della lista oppure viene iterata la lista **eliminando** degli elementi allora conviene utilizzare `LinkedList` poiché vengono eseguite in tempo costante
  - `ArrayList` ha un parametro iniziale (capacità iniziale) che identifica la dimensione iniziale dell'array utilizzato per memorizzare gli elementi
  - `LinkedList` non ha alcun parametro
- Implementazioni di `Map` uguali a quelle di `Set`

Collection (1)

```
public interface Collection<E> extends Iterable<E> {
        int size();
        boolean isEmpty();
        boolean contains(Object o);
        boolean add(E e); // Opt
        boolean remove(Object o); // Opt
        Iterator<E> iterator();
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<? extends E> c); // Opt
        boolean removeAll(Collection<?> c); // Opt
        boolean retainAll(Collection<?> c); // Opt
        void clear(); // Opt
        Object[] toArray();
        <T> T[] toArray(T[] a);
}
```

```
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Altri metodi riguardanti parte funzionale

Collection (2)

- Per **convenzione** tutte le implementazioni hanno un costruttore con argomento una `Collection` che inizializza la nuova collezione con gli elementi di quella specificata

- Esempio
  - Supponiamo di avere una `Collection<String> c` (che può essere un `Set` oppure una `List`)
  - `List<String> l = new ArrayList<>(c);`

Collection (3)

› Esempio 1

```
public class Collections1 {
    public static void main( String[] args ) {
        Collection<String> c = new ArrayList<>();
        c.add( "ten" );
        c.add( "eleven" );
        System.out.println( c );
        Object[] array = c.toArray();
        for ( int i = 0; i < array.length; i++ ) {
            String element = ( String ) array[ i ];
            System.out.println( "Elemento di array:" + element );
        }
        String[] array1 = ( String[] ) c.toArray();  //ClassCastException
        String[] str = ( String[] ) c.toArray( new String[ 0 ] );
        for ( String element: str ) {
           System.out.println( "Elemento di str: " + element );
        }
    }
}
```

Collection (4)

› Esempio 2 (iteratore)

```
public class Collections2 {
    public static void main( String[] args ) {
        Collection<String> c = new ArrayList<>();
        c.add( "ten" );
        c.add( "eleven" );
        for ( Iterator<String> i = c.iterator(); i.hasNext(); ) {
            String element = i.next();
            System.out.println( "Elemento i-esimo:" + element );
            if ("eleven".equals(element) ) {
                i.remove();
            }
        }
        System.out.println( "Numero Elementi: " + c.size() );
    }
}
```

Collection (5)

› Esempio 3 (for each)

```
public class Collections3 {
    public static void main( String[] args ) {
        Collection<String> c = new ArrayList<>();
        c.add( "ten" );
        c.add( "eleven" );
        for ( String element : c ) {
            System.out.println( "Elemento i-esimo:" + element );
            /*
               if ("eleven".equals(element) ) {
                i.remove(); //Non e' possibile rimuovere l'elemento i-esimo con foreach
               }
             */
        }
    }
}
```

Collection (6)

- Prima dell'avvento dei generics i contenitori in Java memorizzavano oggetti di tipo Object e sue sotto-classi
  - `boolean contains(Object element);`
  - `boolean add(Object element);`
  - `boolean remove(Object element);`
- Era necessario effettuare un casting quando si recupera l'oggetto
- Era possibile contenere tipi eterogenei

Collection (7)

› Esempio

```
public class Dog {
  private int dogNumber;
  public Dog(int i) { dogNumber = i; }
  public void id() {
    System.out.println("Dog #" + dogNumber);
  }
}
public class Cat {
  private int catNumber;
  public Cat(int i) { catNumber = i; }
  public void id() {
    System.out.println("Cat #" + catNumber);
  }
}
```

Collection (8)

```
public class CatsAndDogs {
  public static void main(String[] args) {
    List cats = new ArrayList();
    for(int i = 0; i < 7; i++) {
      cats.add(new Cat(i));
    }
    // Nessun problema ad aggiungere un Dog alla lista dei gatti
    cats.add(new Dog(7));

    for(int i = 0; i < cats.size(); i++) {
      ((Cat)cats.get(i)).id();
    }
    //Dog è individuato soltanto a run-time
  }
}
```

List (1)

```
public interface List<E> extends Collection<E> {
        //Positional Access
        E get(int index);
        E set(int index, E element);
        //Optional
        void add(int index, E element);
        E remove(int index);
        boolean addAll(int index, Collection c);
        // Search
        int indexOf(Object o);
        int lastIndexOf(Object o);
        // Iteration
        ListIterator<E> listIterator();
        ListIterator<E> listIterator(int index);
        // Range-view
        List<E> subList(int from, int to);
}
```

List (2)

```
interface ListIterator<E> extends Iterator<E> {
      boolean hasNext();
      E next();
      boolean hasPrevious();
      E previous();
      int nextIndex();
      int previousIndex();

      // Optional
      void remove();
      void set(E o);
      void add(E o);
}
```

Ereditati da Iterator

Ereditato da Iterator

List (3)

- Possono esserci elementi duplicati
- Operazioni aggiuntive
  - Accesso posizionale (si parte da 0) ovvero manipolazione degli elementi in base alla posizione nella lista
  - Ricerca di un determinato oggetto e ritorno della posizione numerica
  - Sotto-liste
  - Implementazioni `ArrayList`, `LinkedList` e `Vector`
  - `add` e `addAll` aggiungono i nuovi elementi alla fine
- Due Liste sono uguali se gli elementi sono gli stessi nello stesso ordine

List (4)

› Esempio 1

```
public class TestList {
    public static void main( String[] args ) {
        List<String> list = new ArrayList<>();
        boolean b;
        Object o;
        int i;
        list.add(1, "x");                    // Aggiunge ad indice 1
        list.add("x");                       // Aggiunge alla fine
        b = list.contains("1");              // E' presente?
```

List (5)

```
……
        //Liste permettono l'accesso causale
        // LinkedList è più costoso che ArrayList
        o = list.get( 1 );                      // Prende oggetto ad indice 1
        i = list.indexOf( "1" );                // Ritorna indice dell'oggetto
        b = list.isEmpty();                     // Lista vuota?
        i = list.lastIndexOf( "1" );            // L'ultimo indice dell'oggetto
        list.remove( 1 );                       // Rimuove oggetto indice 1
        list.remove( "3" );                     // Rimuove oggetto
        i = list.size();                        // Quanti elementi
        list.clear();                           // Rimuove tutti gli elementi
    }
}
```

List (6)

› Esempio 2

```
import java.util.*;
public class Shuffle {
      public static void main(String args[]) {
                  List<String> l = new ArrayList<>();
                  for (int i=0; i<args.length; i++) {
                        l.add(args[i]);
                  }
                  Collections.shuffle(l, new Random());
                  System.out.println(l);
      }
}
```

List (7)

› Esempio 3

```
import java.util.*;
public class Shuffle {
      public static void main(String[] args) {
                  List<String> l = Arrays.asList(args);
                  Collections.shuffle(l);
                  System.out.println(l);
      }
}
```

List (8)

› Esempio completo

```
public class Deal {
          public static void main(String args[]) {
                    int numHands = Integer.parseInt(args[0]);
                    // Make a normal 52-card deck
                    int cardsPerHand = Integer.parseInt(args[1]);
                    String[] suit = new String[] {"spades", "hearts", "diamonds", "clubs"};
                    String[] rank = new String[] {"ace","2","3","4","5","6","7","8",”9","10", "jack","queen","king"};
                    List<String> deck = new ArrayList<>();
                    for (int i=0; i<suit.length; i++)
                              for (int j=0; j<rank.length; j++)
                                        deck.add(rank[j] + " of " + suit[i]);
                    Collections.shuffle(deck);
                    for (int i=0; i<numHands; i++)
                              System.out.println(dealHand(deck, cardsPerHand));
          }
```

List (9)

```
.....
    public static <E> List<E> dealHand(List<E> deck, int n) {
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize-n, deckSize);
        List<E> hand = new ArrayList<>(handView);
        handView.clear();
        return hand;
    }
}

C:> java Deal 4 5
OUTPUT
[ace of clubs, ace of diamonds, 9 of diamonds, 2 of spades, 2 of diamonds]
[7 of hearts, king of hearts, 3 of hearts, 3 of clubs, 2 of clubs]
[jack of clubs, 4 of spades, 9 of spades, 10 of spades, queen of spades]
[4 of clubs, queen of clubs, 9 of hearts, 10 of hearts, 6 of clubs]
```

Set (1)

```
public interface Set<E> extends Collection<E> {
        // Basic Operations
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(E e); //Opt
        boolean remove(Object element); //Opt
        Iterator<E> iterator();
        // Bulk Operations
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<? extends E> c); //Opt
        boolean removeAll(Collection<?> c); //Opt
        boolean retainAll(Collection<?> c); //Opt
        void clear(); //Opt
        // Array Operations
        Object[] toArray();
        <T> T[] toArray(T[] a);
}
```

Set (2)

- Stessi metodi dell'interfaccia `Collection`
- Sono proibiti elementi `e1` ed `e2` tali che `e1.equals(e2)` ed almeno uno è `null` (ovvero duplicati)
- Viene aggiunto un contratto più forte sulle operazioni `equals` ed `hashCode`
- Implementazioni `HashSet TreeSet`
- Esempio
  - Supponiamo di avere una `Collection<String> c`
  - `Collection<String> noDups = new HashSet(c);`
  - **Vengono eliminati i duplicati**

Set (3)

› Esempio 1

```
public class FindDups {
        public static void main(String args[]) {
                Set<String> s = new HashSet<>();
                for (int i=0; i<args.length; i++) {
                    if (!s.add(args[i]))
                         System.out.println("Duplicate detected: "+args[i]);
                }
                System.out.println(s.size()+" distinct words detected: "+s);
        }
}
C:> java FindDups i came i saw i left
OUTPUT
Duplicate detected: i
Duplicate detected: i
4 distinct words detected: [came, left, saw, i]
```

NOTA: Modificando `HashSet` in `TreeSet` si ottiene l'ordinamento

Set (4)

› Esempio 2

```
public class FindDups2 {
        public static void main(String args[]) {
                Set<String> uniques = new HashSet<>();
                Set <String> dups = new HashSet<>();
                for (int i=0; i<args.length; i++)
                        if (!uniques.add(args[i]))
                                dups.add(args[i]);
                uniques.removeAll(dups);
                // Destructive set-difference
                System.out.println("Unique words: " + uniques);
                System.out.println("Duplicate words: " + dups);
   }
}
C:> java FindDups2 i came i saw i left
OUTPUT
Unique words: [came, left, saw]
Duplicate words: [i]
```

Map (1)

```
public interface Map<K,V> {
        // Basic Operations
        V put(K key, V value);
        V get(Object key);
        V remove(Object key);
        boolean containsKey(Object key);
        boolean containsValue(Object value);
        int size(); boolean isEmpty();
        // Bulk Operations
        void putAll(Map<? extends K, ? extends V> m);
        void clear();
        // Collection Views public
        Set<K> keySet();
        Collection<V> values();
        Set<Map.Entry<K, V>> entrySet();
```

……

Map (2)

.....

```
        //Interface for entrySet elements
        interface Entry<K,V> {
                    K getKey();
                    V getValue();
                    V setValue(V value);
        }

}
```

Map (3)

- Oggetto che mappa chiavi a valori
- Non può contenere duplicati delle chiavi
- Implementazioni `HashMap`, `TreeMap`, `LinkedHashMap`, `Hashtable`
- Per **convenzione** tutte le implementazioni hanno un costruttore con argomento una `Map` che inizializza la nuova mappa con gli elementi di quella specificata
- Esempio
  - Supponiamo di avere `Map<String, String> m`
  - `Map<String, String> m1 = new HashMap(m);`

Map (4)

› Esempio 1

```
public class TestMap {
    public static void main( String[] args ) {
        Map<String, String> map = new HashMap<>();
        map.put( "key1", "value1" );
        map.put( "key2", "value2" );
        map.put( "key3", "value1" );
        String value = map.get( "key1" );
        System.out.println( "Valore: " + value );
        System.out.println( "Valore: " + map.get( "key" ) );
    }
}
```

Map (5)

› Esempio 2

```
public class Freq {
        public static void main(String args[]) {
                Map<String, Integer> m = new HashMap<>();
                // Initialize frequency table from command line
                for (String a : args) {
                        Integer freq = m.get(a);
                        m.put(a,(freq==null ? 1 : (freq +1)));
                }
                System.out.println(m.size()+" distinct words detected:");
                System.out.println(m);
        }
}
C:> java Freq if it is to be it is up to me to delegate
OUTPUT
8 distinct words detected:
{delegate=1, be=1, me=1, is=2, it=2, to=3, up=1, if=1}
```

Map (6)

- E' possibile ottenere delle collezioni dalla Mappa
  - `Set<K> keySet()`
    - Insieme delle chiavi contenute nella Mappa
  - `Collection<V> values()`
    - Collezione dei valori contenuti nella Mappa
    - Non è un insieme (`Set`) poiché la mappa può contenere valori multipli ovvero più valori associati alla stessa chiave
  - `Set<Map.Entry<K, V>> entrySet()`
    - L'insieme delle coppie chiave-valore contenute nella mappa
    - Elemento dell'insieme `Map.Entry`

Map (7)

› Esempi

```
for (Iterator<String> i=m.keySet().iterator(); i.hasNext(); ) {
        String s = i.next();
        System.out.println(s);
}
for (Iterator<Map.Entry<String,Integer>> i=m.entrySet().iterator(); i.hasNext(); ) {
                Map.Entry<String,Integer> element = i.next();
                String key = element.getKey();
                Integer value = element.getValue();
                System.out.println(key + "=" + value);
}
for (Map.Entry<String,Integer> element : m.entrySet()) {
                String key = element.getKey();
                Integer value = element.getValue();
                System.out.println(key + "=" + value);
}
```

Queue (1)

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);
    boolean offer(E e);
    E remove();
    E poll();
    E element();
    E peek();
}
```

Type of Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queue (2)

- Collezione utilizzata per mantenere elementi multipli in base ad un ordine
- Tipicamente elementi sono ordinati in FIFO
  - Nuovi elementi sono inseriti alla fine della coda
- Code con priorità elementi sono ordinati in base al loro valore
  - Testa della coda contiene element che può essere rimosso da una chiamata a `remove` o `poll`
- Alcune implementazioni restringono il numero di elementi contenenti della coda (bounded)
  - Implementazioni dentro `java.util.concurrent` sono bounded, in `java.util` no

Queue (3)

- Metodo `add` (ereditato da `Collection`) inserisce un elemento nella coda a meno che non viola le restrizioni di capacita (`IllegalStateException`)
- Metodi `remove` e `poll` rimuovono la testa della coda
  - `remove` eccezione `NoSuchElementException` se coda è vuota `poll` ritorna `null`
- Metodi `element` e `peek` ritornano elemento alla testa della coda
  - `element` eccezione `NoSuchElementException` se coda è vuota `peek` ritorna `null`

Queue (4)

- Implementazioni di Queue generalmente non permettono inserimento elementi `null`
  - `LinkedList` per ragioni di compatibilità si
  - Fare attenzione che `poll` e peek ritornano `null`
- Interfaccia Queue non definisce metodi bloccanti per la coda (es. se la coda è vuota non blocca il chiamante)
  - `java.util.concurrent.BlockingQueue` estende da `Queue` si
- Implementazioni `LinkedList`, `PriorityQueue`

Queue (5)

› Esempio

```
public class Countdown {
 public static void main(String[] args) throws InterruptedException {
      int time = Integer.parseInt(args[0]);
      Queue<Integer> queue = new LinkedList<Integer>();

      for (int i = time; i >= 0; i--)
            queue.add(i);
      while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
      }
    }
}
```

Queue (6)

› Esempio

```
public class PriorityInteger {
    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new PriorityQueue<Integer>();
        Random random = new Random();
        for (int i = 0; i<= time; i++) {
         int nextNumber = random.nextInt();
         System.out.println(nextNumber);
           queue.add(nextNumber);
        }
        System.out.println();
        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

Dequeue (1)

```
public interface Deque<E> extends Queue<E> {
    void addFirst(E e);
    void addLast(E e);
    boolean offerFirst(E e);
    boolean offerLast(E e);
    E removeFirst();
    E removeLast();
    E pollFirst();
    E pollLast();
    E getFirst();
    E getLast();
    E peekFirst();
    E peekLast();
    boolean removeFirstOccurrence(Object o);
    boolean removeLastOccurrence(Object o);
……..……
```

Dequeue (1)

```
.....
    // *** Stack methods ***
    void push(E e);
    E pop();
    // *** Collection methods ***
    boolean remove(Object o);
    boolean contains(Object o);
    public int size();
    Iterator<E> iterator();
    Iterator<E> descendingIterator();
}
```

Type of Operation	First Element (Beginning of the Deque instance)	Last Element (End of the Deque instance)
Insert	`addFirst(e), offerFirst(e)`	`addLast(e), offerLast(e)`
Remove	`removeFirst(), pollFirst()`	`removeLast(), pollLast()`
Examine	`getFirst(), peekFirst()`	`getLast(), peekLast()`

Dequeue (2)

- Generalmente pronunciata *deck*
- Collezione lineare di elementi che supporta inserimento e cancellazione all'inizio e alla fine
- Interfaccia che contiene metodi sia per `Stack` che per *coda*
- Implementazioni `ArrayDeque` e `LinkedList`

Ordinamento (1)

- `List<?> l` può essere ordinata nel seguente modo
  - `Collections.sort(l)`
    - Algoritmo utilizzato: variante merge sort
  - Se gli elementi della lista sono
    - `String` ordinamento lessicografico
    - `Date` ordinamento cronologico
    - .....
  - Ma come è possibile?
    - `String` e `Date` implementano l'interfaccia `Comparable`

```
public interface Comparable<T> {
      int compareTo(T o);
}
```

    - Viene lanciata un'eccezione (`ClassCastException`) se gli elementi non implementano `Comparable` utilizzando il metodo `sort()`

Ordinamento (2)

Implementano Comparable	Descrizione
`Byte`	Numerico con segno
`Character`	Numerico senza segno
`Long`	Numerico con segno
`Integer`	Numerico con segno
`Short`	Numerico con segno
`Double`	Numerico con segno
`Float`	Numerico con segno
`BigInteger`	Numerico con segno
`BigDecimal`	Numerico con segno
`File`	Lessicografico sul path name (dipende dal S.O.)
`String`	Lessicografico
`Date`	Cronologico
`CollationKey`	Lessicografico locale-specific

Ordinamento (3)

```
public class Name implements Comparable<Name> {
    private String  firstName, lastName;
    public Name(String firstName, String lastName) {
        if (firstName==null || lastName==null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String firstName()    {return firstName;}
    public String lastName()     {return lastName;}
    public boolean equals(Object o) {
        if (this == o ) return true;
        if (!(o instanceof Name))
            return false;
        Name n = (Name)o;
        return n.firstName.equals(firstName) && n.lastName.equals(lastName);
    }
```

Ordinamento (4)

```
.....
  public int hashCode() {
        return 31 * firstName.hashCode() + lastName.hashCode();
  }

  public String toString() {
         return firstName + " " + lastName;
  }

  public int compareTo(Name n) {
      int lastCmp = lastName.compareTo(n.lastName);
      return (lastCmp!=0 ? lastCmp : firstName.compareTo(n.firstName));
  }
}
```

Ordinamento (5)

```
class NameSort {
    public static void main(String args[]) {
        Name n[] = {
            new Name("John", "Lennon"),
            new Name("Karl", "Marx"),
            new Name("Groucho", "Marx"),
            new Name("Oscar", "Grouch")
        };
        List<Name> l = Arrays.asList(n);
        Collections.sort(l);
        System.out.println(l);
    }
}
$ java NameSort
OUTPUT
[Oscar Grouch, John Lennon, Groucho Marx, Karl Marx]
```

Ordinamento (6)

› Interfaccia

```
public interface Comparator<T> {
     int compare(T o1, T o2);
}
```

› Maggior controllo sull'ordinamento

› Permette l'ordinamento su oggetti che **non** implementano l'interfaccia `Comparable`

› E' possibile effettuare diversi ordinamenti mediante tale interfaccia

› E' simile all'interfaccia `Comparable` soltanto che il confronto viene effettuato al di fuori della classe

Ordinamento (7)

› Esempio

```
public class Employee implements Comparable<Employee> {
    private Name name;
    public Name name() {return name;}
    public int employeeNumber() {……}
    public Date hireDate() {……}
    public int compareTo(Employee eR) {
      return name.compareTo(eR.name);
    }
……
}
```

Ordinamento (8)

```
public class EmpSort {
        static SeniorityOrder SENIORITY_ORDER = new SeniorityOrder();
        // Employee database
        static final Collection<Employee> employees = ... ;
        public static void main(String[] args) {
                        List<Employee> e = new ArrayList<>(employees);
                        Collections.sort(e, SENIORITY_ORDER);
                        System.out.println(e);
        }
}
class SeniorityOrder implements Comparator<Employee> {
        public int compare(Employee e1, Employee e2) {
                return e2.hireDate().compareTo(e1.hireDate());
        }
}
```

SortedSet (1)

- `SortedSet` è un `Set` dove i suoi elementi sono mantenuti in ordine ascendente utilizzando l'interfaccia `Comparable` oppure `Comparator` (fornita all'atto di creazione)
- Per **convenzione** le implementazioni vengono fornite di un costruttore che ha come argomento
  - Una `Collection`
  - Un `SortedSet`
  - Un `Comparator`
  - Un `Comparator` e un `SortedSet`

SortedSet (2)

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    // Endpoints
    E first();
    E last();
    // Comparator access
    Comparator<? super E> comparator();
}
```

- `subSet()` estremo sx escluso ed estremo dx incluso
- `headSet()` dall'inizio fino all'elemento specificato escluso
- `tailSet()` dall'elemento specificato fino alla fine

SortedMap (1)

- `SortedMap` è un `Map` dove i suoi elementi sono mantenuti in ordine ascendente utilizzando l'interfaccia `Comparable` oppure `Comparator` (fornita all'atto di creazione)
- Per **convenzione** le implementazioni vengono fornite di un costruttore che ha come argomento
  - Una `Collection`
  - Un `SortedMap`
  - Un `Comparator`
  - Un `Comparator` e un `SortedMap`

SortedMap (2)

```
public interface SortedMap<K, V> extends Map<K, V> {
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

› `subMap()`, `headMap()`, `tailMap()` come `SortedSet`

Implementazioni Wrapper (1)

- Sono implementazioni che delegano il loro reale lavoro ad una specifica collezione aggiungendo funzionalità extra in cima alle collezioni esistenti (decorator pattern)
- Tali implementazioni si trovano all'interno della classe di utility `Collections` che fornisce soltanto metodi statici
- Esempio
  - `Collection` sono tutte non sincronizzate (thread-unsafe)
  - E' possibile sincronizzarle mediante metodi contenuti in `Collections`

Implementazioni Wrapper (2)

- Synchronization

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

- Unmodificable

```
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);
public static <T> Set<T> unmodifiableSet(Set<? extends T> s);
public static <T> List<T> unmodifiableList(List<? extends T> list);
public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);
public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<? extends T> s);
public static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> m);
```

Altri metodi di Collections

› Ricerca binaria

```
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
```

› Shuffle: randomizza la collection

```
public static void shuffle(List<?> list)
public static void shuffle(List<?> list, Random rnd);
```

› Other

```
public static void reverse(List<?> list);
public static <T> void fill(List<? super T> list, T obj)
public static <T> void copy(List<? super T> dest, List<? extends T> src)
public static <T extends Object & Comparable<? super T>> T min( Collection<? extends T> coll)
public static <T extends Object & Comparable<? super T>> T max( Collection<? extends T> coll)
```