Esercitazione:
Interfaccia Comparable
Linked List

Confronto tra oggetti

Confrontare oggetti ha un significato che dipende dal tipo di oggetto. La classe che definisce l'oggetto deve anche definire il "significato" del confronto.

Se consideriamo la classe `String`, essa definisce il metodo `compareTo` che attribuisce un significato ben preciso all'ordinamento di stringhe: l'ordinamento lessicografico.

Lo stesso vale per tutti i tipi di base `Integer`, `Double`,...

Confronto tra oggetti

Per i tipi di dati primitivi il confronto è ovvio, ma per oggetti generici è necessario definire un apposito criterio di confronto.

Algoritmi e strutture dati che utilizzano tali oggetti sono legati alla specifica implementazione, anche per criteri generali ed astratti come quello della confrontabilità.

Interfaccia Comparable

Per definire un comportamento astratto si usa in Java la definizione di **interfaccia**, che deve essere realizzata (**implementata**) da una classe che dichiari di avere tale comportamento.

L'interfaccia `Comparable` permette di gestire il confronto, se necessario, fra oggetti di qualsiasi tipo.

In particolare nel caso delle strutture dati astratte, consente di implementare le classi e i metodi relativi alla struttura dati scelta senza fare riferimento al tipo degli oggetti da inserire nella struttura stessa.

Interfaccia Comparable

Ad esempio, se un array o una lista contiene oggetti di tipo **Person**, costituiti da un campo stringa per il nome ed un campo intero per l'età, non è ovvio che cosa significhi che un oggetto è minore di un altro.

L'ordinamento tra i vari oggetti può essere fatto in base al campo nome, all'età o con una combinazione dei due.

Nella definizione che segue supporremo che l'ordinamento tra oggetti **Person** avvenga in base al campo **name**.

Interfaccia Comparable

Per rendere confrontabili due oggetti, la classe che li definisce deve implementare l'interfaccia `java.lang.Comparable`, contenente la firma del metodo `compareTo`.

```
public interface Comparable {

        public int compareTo(Object o)

}
```

Interfaccia Comparable

`public interface Comparable`

*"This interface imposes a **total ordering** on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's **compareTo** method is referred to as its natural comparison method".*

`public int compareTo(Object o)`

> *"Compares this object with the specified object for order. Returns a **negative integer**, **zero**, or a **positive integer** as this object is less than, equal to, or greater than the specified object."*

Interfaccia Comparable

I sottotipi di `Comparable` forniscono un metodo per determinare la relazione di ordinamento (devono valere cioè le proprietà **riflessiva**, **antisimmetrica** e **transitiva**) fra i loro oggetti.

L'ordinamento deve essere totale.

Inoltre:

`x.compareTo(y)==0` implica `x.equals(y)`

Interfaccia Comparable: Eccezioni

`int CompareTo(Object o)`

se `o==null`, viene generata `NullPointerException`;

se `this` e `o` non sono confrontabili, viene generata `ClassCastException`;

Classe Person

```
public class Person implements Comparable
{ private String name;
  private int age;

  public Person()
  { this (“ ”, 0);
  }

  public Person(String str, int a)
  { name = str;
    age = a;
  }
…
```

Classe Person

```
  public int compareTo(Object p)
  { return name.compareTo( ((Person) p).name );
  }

  public String toString()
  { return name + " " + age;
  }

  public boolean equals(Object p)
  { if (p == null)
      throw new NullPointerException();
    return name.equals((Person) p).name;
  }

}
```

Esempio

Realizzare una classe **Abbonato** che implementi l'interfaccia **Comparable** ordinando gli abbonati in base al nominativo (cognome/nome).

Chiedere all'utente di inserire alcuni nomi di abbonati, con i relativi numeri di telefono e generare degli oggetti **Abbonato**. Tramite il metodo **compareTo** individuare il primo abbonato e stamparne i dati.

Esempio

```
class Abbonato implements Comparable {
    private String fname, surname;
    private int tel;

    public Abbonato(String n,String c, int t){
        fname = n;
        surname = c;
        tel =t;
   }
    public String toString() {
       return    "Abbonato[cognome="+surname+",    nome    =
   "+fname+" Tel. = "+tel+"]";
   }
    public int compareTo(Object other){
        Abbonato b = (Abbonato)other;
        int i=surname.compareTo(b.surname);
        if (i==0) return fname.compareTo(b.fname);
               else  return i;
   }
}
```

Esempio

```
...
  while (more){
    System.out.println("Inserisci il nome di un abbonato
                                    (QUIT per finire):");
    nome = console.readLine();
    if (nome.toUpperCase().equals("QUIT")) more = false;
       else {
          System.out.println("Inserisci il cognome:");
          c = console.readLine();
          System.out.println("Inserisci il numero di
                                              telefono:");
          telefono = console.readInt();
         Abbonato current =new Abbonato(nome, c, telefono);

          if (first== null || first.compareTo(current)>0)
              first = current;
        }
    }
     System.out.println("Il primo abbonato è: "+first);
  }
}
```

Interfacce & Tipi di Dati

Nell'uso delle interfacce in un programma, valgono alcune semplici regole:

Possiamo dichiarare una variabile indicando come tipo un'interfaccia:

```
Comparable cmp;
```

Non possiamo istanziare un'interfaccia:

```
Comparable cp = new Comparable(); // VIETATO
```

Interfacce & Tipi di Dati

Ad una variabile di tipo interfaccia possiamo assegnare solo istanze di classi che implementano l'interfaccia:

```
Comparable cp = new Person();
```

Queste conversioni tra un oggetto ed un riferimento ad una delle interfacce che sono “realizzate” dalla sua classe sono automatiche. L’interfaccia assume il ruolo di una “Super classe”.

Su di una variabile di tipo interfaccia possiamo invocare solo metodi dichiarati nell'interfaccia (o nelle sue "super-interfacce").

Esempi

```
...
Comparable stuff, stuff1;       // OK
stuff = new Comparable();       // NO!!
stuff = new Integer(5);         // OK
stuff1 = "pippo";                 // OK

System.out.print(stuff.intValue());  // NO!!

System.out.print(stuff.compareTo(stuff1));
//OK, compila correttamente.
 ...
```

Liste generiche

Possiamo quindi costruire strutture dati generiche il cui campo **info** del nodo della struttura sia di tipo generico **Object**.

L'ordinamento degli elementi è quindi determinato dall'algoritmo presente nel metodo **CompareTo()** richiamato sugli elementi **info**.

Liste generiche

```
public class Nodo
{ protected Object info; // elemento generico
  private Nodo next;

  public Nodo(Object x)
  {  this(x, null);
  }

  public Nodo(Object obj, Nodo n)
  { info = obj;
    next = n;
  }
…
```

Liste generiche

```
public Nodo searchOrd( Object obj )
{ Nodo aux = head;
  for( ; aux != null &&
          ((Comparable)aux.getInfo()).compareTo(obj) < 0;
          aux = aux.getNext() ) ;
  if (aux != null && (aux.getInfo()).equals(obj) )
       return aux;
  return null;
}
```

Esercizio

Esercizio:

Definire in maniera completa la classe **Lista** per una lista linkata semplice di oggetti generici.

Testare la struttura per memorizzare elementi della classe **Person**.

Esercizi

1. Scrivere un metodo per verificare se due liste semplicemente concatenate hanno lo stesso contenuto (nello stesso ordine).

2. Concatenare una lista semplicemente concatenata ad un'altra lista semplicemente concatenata.

3. Rimuovere da una lista semplicemente concatenata un elemento di posto dato.

4. Fondere due liste semplicemente concatenate ed ordinate di interi in una lista semplicemente concatenata ordinata.

Esercizio 1

```
public static boolean compareLists(LinkedList l1,LinkedList
   l2) {

   if (l1.isEmpty() & l2.isEmpty()) return true;
   if (l1.isEmpty() | l2.isEmpty()) return false;

   ListNode pl1=l1.getHead();
   ListNode pl2=l2.getHead();

   while(pl1!= null && pl2!= null){
   if(((Comparable)pl1.getInfo()).compareTo(pl2.getInfo())!=0)
          return false;
       else{
         pl1=pl1.getNext();
         pl2=pl2.getNext();
       }
   }
   if (pl1== null && pl2== null) return true;
      else return false;
  }
```

Esercizio 2

```
public   static   void   concatLists(LinkedList   l1,
  LinkedList l2){

  if (l2.isEmpty()) return;

  ListNode pl2=l2.getHead();
  for(; pl2!=null; pl2=pl2.getNext())
    l1.insertTail(pl2.getInfo());

 }
```

Esercizio 3

```
public ListNode deleteAt(int pos){

      ListNode aux = head;
      ListNode pred = null;

     for(int count = 0; aux != null && count != pos;
         pred = aux, aux = aux.next, count++);

         if (aux != null){
           if (pred == null)
                  head = head.next;
           else
                  pred.next = aux.next;
           size --;
          }
         return aux;
        }
```

Esercizio 4

```
public    static    LinkedList    mergeList(LinkedList    l1,
  LinkedList l2) {
    LinkedList l3 = new LinkedList();
    ListNode p1 = l1.getHead();
    ListNode p2 = l2.getHead();

    while(p1!= null && p2!= null) {
       int temp =
       ((Comparable) p1.getInfo()).compareTo(p2.getInfo());

     if (temp <= 0 )   {
          //l'elemento corrente di l1 è <= a quello di l2
         l3.insertTail(p1.getInfo());
         p1 = p1.getNext();       }
      else    {
         //l'elemento corrente di l1 è > a quello di l2
         l3.insertTail(p2.getInfo());
         p2 = p2.getNext();        }
     }//End while
```

Esercizio 4

```
...
// Attacchiamo l'eventuale coda della lista piu' lunga

while(p1!= null) {
       // l1 e' la lista piu' lunga
         l3.insertTail(p1.getInfo());
         p1 = p1.getNext();
     }

    while(p2!= null)     {
       // l2 e' la lista piu' lunga
         l3.insertTail(p2.getInfo());
         p2 = p2.getNext();
     }
    return l3;
  }
```

Esercizio 5

Invertire una lista semplicemente concatenata mediante una procedura ricorsiva.

Lo faremo in seguito….

Esercizi

Esercizio 6 (8.4 – pag 163 Hubbard) Scrivere un metodo:

```
public static void exchange(LinkedList list, int i, int j)
```

che inverta tra loro gli elementi i-esimo e j-esimo.

Esercizio 6

```
public static void exchange (LinkedList l1,int i, int j)
   {
     int cont=0;
     if (l1.isEmpty()) return;
     ListNode pl1=l1.getHead();
     ListNode aux1=null, aux2=null;
      for(; pl1!=null; pl1=pl1.getNext()){
          cont++;
          if (cont==i) aux1=pl1;
          if (cont==j) aux2=pl1;
      }
    if ((aux1!=null) && (aux2!=null)) {
        Object p1=aux1.getInfo();
        aux1.setInfo(aux2.getInfo());
        aux2.setInfo(p1);
     }
   }
```

Esercizi

Esercizio 7 – Il problema di Josephus (Esempio 8.6 pag 160 – Hubbard) – Si basa sul racconto dello storico Josephus circa l'esito di un patto suicida da lui stipulato con altri 40 soldati, finiti in un imboscata dei romani nel 67 a.C. Josephus propose ad ogni uomo di uccidere il suo vicino ed egli, furbamente, fece in modo di rimanere per ultimo, così da sopravvivere per raccontare la vicenda.

Simulare questo problema utilizzando una lista semplicemente concatenata.

(Variante Esercizio 8.5 pag 163)

Esercizio 7

```
public static void main(String[] args) {
    LinkedList l1 = new LinkedList( );
    int N=(int)((Math.random()*10)+1);

  System.out.println("Il numero di soldati e': "+N);
    for (int k=0; k<N; k++){
        char soldier[]={(char)('A'+k)};
        String p= new String(soldier);
        l1.insertTail(p);
      }
    l1.printList();
…
```

Esercizio 7

```
…
ListNode pl2=l1.getHead();
while(l1.getSize()>1)
  {
   if (pl2.getNext()==null){
         System.out.println(pl2.getInfo()+"uccide"+
                                    l1.getHead().getInfo());
         l1.deleteHead();}
   else {
         System.out.println(pl2.getInfo()+"uccide"+
                                   pl2.getNext().getInfo());
         l1.deleteKey((pl2.getNext()).getInfo());}
   pl2=pl2.getNext();
   if (pl2==null) pl2=l1.getHead();
    }
System.out.println("Il solo sopravvissuto e': ");
l1.printList();
    }
```

Esercizi

Esercizio 8 – (Num. 11 – pag 124 Drozdek) Scrivere un metodo per invertire una lista semplicemente concatenata usando una sola scansione attraverso la lista.

IDEA:

Scorrere la lista e dal secondo elemento in poi, eseguire una cancellazione del nodo seguita da un inserimento in testa.

Esercizio 9 – (Num. 16 – pag. 124 Drozdek) Inserire un nodo esattamente al centro di una lista doppiamente concatenata.

IDEA: Scandire la lista sia dalla testa che dalla coda, tramite due nodi di supporto. Appena i due riferimenti coincidono abbiamo raggiunto il centro (bisogna solamente distinguere i 2 casi in cui abbiamo un numero complessivo di elementi pari o dispari)