Gerarchie e polimorfismo: liste

Generalizzare le liste di interi

- ✓ `List`
- ✓ lista di oggetti
  - non modificabile
- ✓ vorremo poi definire un sottotipo
  - versione ordinata

List

- classe astratta
- usate i sottotipi per implementare i due casi della definizione ricorsiva
  - lista vuota
  - lista non vuota

Specifica del supertipo `List`

```
public abstract class List {
 // OVERVIEW: un List è una lista non modificabile di Objects.
 // Elemento tipico [x1,...,xn]
 public abstract Object first () throws EmptyException;
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  // ritorna il primo elemento di this
 public abstract List rest () throws EmptyException;
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  // ritorna la lista ottenuta da this togliendo il primo elemento
 public abstract Iterator elements ();
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 // this (come Objects) nell'ordine che hanno in this
 public abstract List addEl (Object x);
  // EFFECTS: restituisce la lista ottenuta aggiungendo x all'inizio di this
 public abstract List remEl (Object x);
  // EFFECTS: restituisce la lista ottenuta rimuovendo x da this
 public abstract int size ();
  // EFFECTS: ritorna il numero di elementi di this
 public abstract boolean repOk ();
 public String toString ();
 public boolean equals (List o);
}
```

Implementazione del supertipo `List`

```
public abstract class List {
 // OVERVIEW: un List è una lista non modificabile di Objects.
 // Elemento tipico [x1,...,xn]
 // metodi astratti
 public abstract Object first () throws EmptyException;
 public abstract List rest () throws EmptyException;
 public abstract Iterator elements ();
 public abstract List addEl (Object x);
 public abstract List remEl (Object x);
 public abstract int size ();
 public abstract boolean repOk ();
 // metodi concreti
 public String toString () {....}
 public boolean equals (List o) {.... }
}
```

- ✓ implementare `toString` e `equals`
  - utilizzando il generatore `elements`

Implementazione del sottotipo `EmptyList`

```
public class EmptyList extends List {
 public EmptyList () {}
 public Integer first () throws EmptyException {....}
 public List rest () throws EmptyException {....}
 public Iterator elements () { return new EmptyGen(); }
 public List addEl (Object x) {....}
 public List remEl (Object x) {....}
 public int size () {....}
 public boolean repOk () {....}
 static private class EmptyGen implements Iterator {
   EmptyGen () {}
   public boolean hasNext () { return false; }
   public Object next () throws NoSuchElementException {
        throw new NoSuchElementException("List.elements"); }      }

}
```

Implementazione del sottotipo `FullList`

```
public class FullList extends List {
 private int sz;
 private Object val;
 private List next;
 public FullList (Object x)
 {sz = 1; val = x; next = new EmptyList (  ); }
 public Integer first () throws EmptyException {....}
 public List rest () throws EmptyException {....}
 public Iterator elements () {....}
 public List addEl (Object x) {....}
 public List remEl (Object x) {....}
 public int size () {....}
 public boolean repOk () {....}}
```

Il sottotipo `OrderedList`

```
public class OrderedList extends? List {
 // OVERVIEW: una OrderedList è un sottotipo di List, che ha una operazione in più
 //  per inserire un elemento che tiene conto dell'ordine
 public OrderedList ();
 // EFFECTS: restituisce la lista ottenuta inserendo x in this
 public OrderedList addEl (Comparable x);
  // EFFECTS: restituisce la lista ottenuta inserendo x in this in modo che il
  // risultato sia una lista ordinata. Solleva leveccezioni che deve
}
```

- rifarlo anche con sottotipi di `Comparator`

Il sottotipo `OmoList`

- rifare la gerarchia partendo da una versione di `List` che garantisca l'omogeneità
- suggerimento: usare nella rappresentazionme

```
private Class type;
```

- e poi controllare sempre che i tipi siano omogenei

```
type = x.getClass();

type.isInstance(y)
```