`LRStruct`: An Enhanced `AList`

- `LRStruct` supports additional operations on the list structure:
  - `void insertFront(Object dat)`
  - `Object removeFront()`
  - `void setFirst(Object dat)`
  - `void setRest(LRStruct tail)`

Contrasting `LRStruct` from `AList`

- Constructing an `AList` of one object ...

```
{ ...
  AList list = ListFactory.Singleton.makeEmptyList();
  list = ListFactory.Singleton.makeNEList(new Integer(-1),
                                          list);
  ...
```

- Constructing an `LRStruct` of one object ...

```
{ ...
  LRStruct list = new LRStruct();     // an empty list
  list.insertFront(new Integer(-1));
  ...
```

Contrasting `LRStruct` from `AList` (cont.)

- What is printed?

```
{ ...
  AList list = ListFactory.Singleton.makeEmptyList();
  AList altL = list;
  list = ListFactory.Singleton.makeNEList(new Integer(-1),
                                          list);
  System.out.println(altL);
  ...
```

Contrasting `LRStruct` from `AList` (cont.)

- What is printed?

```
{ ...
  LRStruct list = new LRStruct();      // an empty list
  LRStruct altL = list;
  list.insertFront(new Integer(-1));
  System.out.println(altL);
  ...
```

The State Pattern

- Define an abstract class for the states of the system.
  - This abstract state class should provide all the abstract methods for all of the concrete subclasses.
- Define a concrete subclass of the abstract class for each state of the system.
  - Each concrete state must implement its own concrete methods.
- Represent the system by a class containing an instance of a concrete state.
  - This instance represents the current state of the system.

The State Pattern (cont.)

- Define methods for the system to return the current state and to change state.
- Delegate all requests made to the system to the current state instance.
  - Since this instance can change dynamically, the system will behave as if it can change its class dynamically.