Static Class Members and Singletons

Corky Cartwright
Department of Computer Science
Rice University

DrJava Intermediate Level

- For HW8, we will progress to the Intermediate Language Level. Beware of the fact that Java language levels are not fully upward compatible. DrJava compiles each file based on its file type, but If you generate a `.dj0` test class and manually save it as a `.dj1` test class, it will **not** compile without modification.
- At the Intermediate level, the **`static`**, **`public`**, and **`private`** visibility attributes for classes and methods are enabled, but they are prohibited at the Elementary level. The JUnit framework requires that test classes be **`public`**. The translator for the Elementary level generates the **`public`** attribute for test classes. The Intermediate level does not generate the **`public`** attribute for test classes, but includes it in the provided template.
- The Elementary and Intermediate levels make the same distinction with regard to **`import`** statements.

`static` Class Members

- Almost all of the fields and methods that we have seen thus far have been attached to Java objects (class instances), but fields and methods can also be attached to Java classes. Such fields and methods and called **`static`** class members.
- We will defer discussing **`static`** methods. They are not supported at the Intermediate Level in DrJava. Starting with HW8, set your language level to the Intermediate Level.
- **`static`** fields are used primarily to store constants associated with a class. Why **`static`**? We only need one copy of a constant. It is wasteful to create a copy in every object of a class. You have already seen a few **`static`** fields in the context of Java libraries. The fields **`MAX_VALUE`** and **`MIN_VALUE`**, which are present in all of the wrapper classes except **`Boolean`**, are **`static`**.

`private` Class Members

- Any `static` or dynamic (instance) field or method can be marked as `private`. A `private` field is *visible* only within the class in which it is defined. We use `private` much like Scheme `local` but confining a variable's scope to a class is much less restrictive that confining it to an expression. We will defer discussing `static` methods until later in the course; they are not very important.
- `private` members are used primarily for methods and fields that only concern the class containing them, *e.g*. help methods. Note that in the context of the composite pattern, we cannot make a help method `private`, because the method must be visible in all of the classes in the composite hierarchy.

The Singleton Pattern

An important application of the **`static`** and **`private`** attributes is the *singleton pattern*. Each execution of the expression

`new EmptyIntList()`

creates a new object. In principle, there is only one empty list, just like there is only one number 0. Hence, we would like to represent the empty list by a single object.

Implementing Singleton

A unique instance of a class (*singleton pattern*) can be created using two chunks of code:

- a **`static`** field in the class that holds the single instance of the class
- a **`private`** attribute on the class constructor, so no client can create another instance of the class.

Singleton `IntList`

```
abstract class IntList {
  abstract IntList sort();
  IntList cons(int n) { return new ConsIntList(n, this); }
  abstract IntList insert(int n);
}

class EmptyIntList extends IntList {
  static EmptyIntList ONLY = new EmptyIntList();
  private EmptyIntList() { }
  IntList sort() { return this; }
  IntList insert(int n) { return cons(n); }
}

class ConsIntList extends IntList {
  int first;
  IntList rest;
  IntList sort() { return rest.sort().insert(first); }
  IntList insert(int n) {
    if (n <= first) return cons(n);
    else return rest.insert(n).cons(first);
  }
}
```

Static member holding the unique instance

Private constructor

For Next Class

- Labs this afternoon and tomorrow
- Easy Homework due Friday
- Reading: OO Design Notes, Ch. 1.6-1.8.