Static Class Members and Singletons

Corky Cartwright
Department of Computer Science
Rice University

Review

- Primitive types: `int long short byte double float char boolean`
  - For every primitive type, there is a corresponding wrapper class

`int`	`Integer`
`long`	`Long`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`
`short`	`Short`
`byte`	`Byte`

  - java will generally wrap (box) / unwrap (box) values/objects of primitive/wrapper type. Exception: Java does not unbox wrapper objects if the type of the expression is larger than the wrapper type, *e.g.*, `Object`

Examples: see demo

Review cont.

- Java is much more idiosyncratic than Scheme. If you are confused about some aspect of Java notation, please ask a course staff member or a more experienced classmate. You can also look it up in the online references listed in the course wiki (or by doing a web search). With a bit of practice, Java notation will become familiar (if occasionally puzzling).
- The DrJava functional language level hides some but not all of the grief.
- Read the class notes on OO design and do all of the finger exercises.

DrJava Language Levels

In this lecture, we will use a larger subset of the DrJava functional language level. The names of functional language level files end in the file extension `.dj`. You can also use the legacy file extensions `.dj0` and `.dj1` which can be useful if you want to maintain several versions of the same file. DrJava will ask you if you want to save such such files using the newer `.dj`. file extension.   If you want to retain the legacy file extension, answer “No”.

New constructs:

The `static`, `public`, and `private` visibility attributes for classes and methods. By default, Java classes and methods have “default” (also called “package”) visibility. Since we are putting all of classes in the default (also called “empty”) package, there is no reason for us to use the `public`T visibility attribute with one exception. The JUnit framework requires that test classes be `public`.   The functional language level adds the `public` attribute for test classes if it not already present.

`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.
- `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.

`Private` help methods cannot be tested from a separate class BUT they can be tested using test code within the method's class. We will not use this feature of JUnit (@test annotations) in this class but you will probably encounter it in more advanced courses. Hence, we do not recommend using private methods (except constructors) in this class.

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

- Exam due Wednesday in class
- Optional Homework 6 due at midnight Wed (technically Thursday)
- Lab tomorrow
- Easy Homework due Monday
- Reading: OO Design Notes, Ch. 1.6-1.8.