CMPSCI 187: Programming With Data Structures

Lecture 7: Java: Inheritance and Class Hierarchies
21 September 2011

Java: Inheritance and Class Hierarchies

- Classes Extending Classes: Subtypes and Supertypes
- The Hierarchy of Classes: When is a Reference Legal?
- Adding Interfaces to the Hierarchy
- The `Object` Class: The `toString` and `equals` methods
- Visibility: `public`, `protected`, and `private`
- Inheritance in Project #2

Classes Extending Classes: Subtypes, Supertypes

- When we declare a class, we can have it **extend** another class, as with: `public class Terrier extends Dog.`
- This makes Terrier a **subclass** of Dog, and Dog a **superclass** of Terrier. If CairnTerrier extends Terrier, it is also a subclass of Dog, and so forth.
- Every Terrier object then **is-a** Dog object. Terriers have all the fields of Dogs and can run all their methods. They may also have their own fields and methods, declared in the Terrier declaration. If a Terrier method and a Dog method have the same name, the former **overrides** the latter.
- As we discussed earlier, when we create a Terrier and use it in a Dog variable, it remains a Terrier and when we call an overridden method we get the Terrier version.

The Hierarchy of Classes

- We can draw a diagram (a **graph**) where points (**nodes**) represent classes and we have a line (**edge**) from A down to B whenever B extends A.
- The top of this diagram is the class `Object` because every class extends `Object` by default, with no declaration.
- Since a class can extend only one other class, this graph is a special kind called a **tree**. We use family language for this -- if both B and C extend A, then A is their **parent** and B and C are **siblings**.
- Any two classes have a common ancestor, even if it is only `Object`. In our example, the common ancestor of CairnTerrier and Retriever would be Dog.

Is This Reference Legal?

- Every variable in Java has a type, and the compiler checks types carefully.
- We could assign a Terrier variable to a Dog, but not a Dog to a Terrier.
- If a method expects a Dog as a parameter, we could give it a Terrier. But then the method code would have to work on any Dog, not just Terriers.
- As we saw before, if we know at run time that the contents of a variable really belong to a subtype of the variable's type, we can do a **cast**.
- If d is a Dog variable and we say `Terrier t = (Terrier) d;` this will work if the value of d is really a Terrier at run time. If not, we get a ClassCastException at run time.

Adding Interfaces to the Hierarchy

- An interface, as a Java entity, is a list of methods with signatures.
- In order to implement the interface, a class must have implementations of all those methods and declare that it does so.
- For example, if GuardAnimal is an interface containing the void method patrol, we could define a class with the line `public class Rottweiler extends Dog implements GuardAnimal` as long as the new class contains a `patrol` method.
- We can then draw GuardAnimal in the class inheritance diagram *above* Rottweiler, next to Dog. We draw a dotted line from it down to Rottweiler.
- The new diagram is no longer a tree, as a class can have multiple “parents”.

The `Object` Class: `toString` and `equals`

- As we said, the Object class is the ancestor of all other classes, since every other object is-a Object.
- Object has two methods, which are thus inherited by all other objects: `String toString( )` and `boolean equals (Object x)`.
- The toString method of Object gives a String that denotes the address at which the Object is stored. Normally when you write a class you write a more useful toString method that tells you what you want to know about the contents.
- The equals method of Object tells whether x is *the same object* as the calling Object. This is also the result of using == on objects. If you have a better definition of two objects in a class being “equal”, you write it into a method.

Visibility: `public`, `private`, and `protected`

- We have normally declared every field and method we write to be either `public`, readable and usable by any method in any class, or `private`, which means readable and usable only by methods from the same class.

- The standard is to make all fields private and then write public get and set methods if you want them. This allows you to change the implementation of the field without worrying about other classes interacting with the field directly.

- L&C incorrectly say in section 13 of appendix B that private fields and methods are not inherited. The fields are inherited but cannot be modified by the subclass' code. If you make a field or method `protected`, though, it can be used by any subclass of its class as well as by its class. A protected field or method is still private to any class that is not a subclass of its class.

Inheritance in Project #2

- In Project #2 we need every cell of our maze to carry a boolean that tells whether it has been seen in the search. But Cell objects don't have that field.
- If we make a new class SCell extending Cell, we can give it the new field and SCell objects will retain the fields and methods of the Cell class.
- Unfortunately we now have to change the code of the Maze class to make the array of SCells rather than Cells. (We could make a new SMaze class but it would not necessarily extend Maze and we don't need the old Maze anyway.) For example, we will need a moves method that always returns SCell arrays.
- The right way to have done this, perhaps, would have been to make Maze a generic class, so we would have had Maze<Cell> in Project #1 and Maze<SCell> in Project #2. But we had enough to deal with in Project #1!