Inheritance 2 Lecture Guide | CS 61B Spring 2018

sp18.datastructur.es/materials/lectures/lec9/lec9

Lecture Code

Code from this lecture available at

https://github.com/Berkeley-CS61B/lectureCode-sp18/tree/master/Inheritance2.

Overview

The Interface and implements. Earlier we went classes and interfaces and we realized that when writing classes, we can sometimes write a lot of redundant code. This leads us to Inheritance, the idea that some object does not need to redefine all of its qualities of its parent. We can inherit from both interfaces and classes and the syntax is slightly different. For classes to inherit the qualities of an interface the syntax is as follows (where SLList is a class and List61B is an interface):

```
SLList<Blorp> implements List61B<Blorp>
```

Similarly, the way for a class to implement the qualities of another class the syntax is as follows:

```
Class_Name extends Class_Name
```

Usage of Inheritance. Say we wanted to make a special type of `SLList` called `RotatingSLList`. `RotatingSLList` should be able to do everyhthing that SLList can; however, it should also be able to rotate to the right. How can we do this? Well this is just an application of Inheritance! Doing the following will allow for RotatingSLList to have all the qualities of SLList as well as its own method `rotateRight`.

```
public class RotatingSLList<Blorp> extends SLList<Blorp>{
  public void rotateRight() {...}
}
```

What is Inherited? We have a powerful tool in Inheritance now; however, we will define a few rules. For now, we will say that we can inherit:

- instance and static variables
- all methods
- all nested classes This changes a little bit with the introduction of private variables but don't worry about that right now. The one item that is not inherited is a class's constructor.

The Special Case of the Constructor? Even though constructor's are not inherited, we still use them. We can call the constructor explicitly by using the keyword `super()`. At the start of every constructor, there is already an implicit call to its super class`s constructor. As a result

```
public VengefulSLList() {
  deletedItems = new SLList<Item>();
}
```

is equivalent to

```
public VengefulSLList() {
  super();
  deletedItems = new SLList<Item>();
}
```

However, constructor`s with arguments are not implicitly called. This means that.

```
public VengefulSLList() {
    super(x);
    deletedItems = new SLList<Item>();
  }
```

is not equivalent to

```
public VengefulSLList() {
    deletedItems = new SLList<Item>();
  }
```

This is because only the empty argument `super()` is called.

Is A. When a class inherits from another, we know that it must have all the qualities of it. This means that `VengefulSLList` is a `SLList` because it has all the qualities of an `SLList` - it just has a few additional ones too.

Every single class is a descendent on the Object class, meaning they are all Objects.

Abstraction As you'll learn later in this class, programs can get a tad confusing when they are really large. A way to make programs easier to handle is to use abstraction. Basically abstraction is hiding components of programs that people do not need to see. The user of the hidden methods should be able to use them without knowing how they work.

An intuitive way to realize the motivation of abstraction is to look at yourself. You are a human (unless some robot is looking at this in which case I am sorry for offending you) and humans can eat food and convert it to energy. You do not need to know how you convert food to energy you just know that it works. In this case think of your conversion of food to energy as a method and the input is food and the output is energy.

Casting In Java, every object has a static type (defined at compile-time) and a dynamic type (defined at run-time). Our code may rely on the fact that some variable may be a more specific type than the static type. For example if we had the below definitions:

```
Poodle frank  = new Poodle("Frank", 5);
Poodle frankJr = new Poodle("Frank Jr.", 15);
```

This statement would be valid

```
Dog largerDog = maxDog(frank, frankJr);
```

But this one would not be

```
Poodle largerPoodle = maxDog(frank, frankJr);
```

The reason the former statement is valid is because the compilers knows for a fact that anything that is returned from a `maxDog` function call is a `Dog` . However, in the latter case, the compiler does not know for a fact that the return value of `maxDog` would result in a `Poodle` even though both `Dog` arguments are `Poodle` s.

Instead of being happy with just having a generic `Dog` , we can be a bit risky and use a technique called casting. Casting allows us to force the static type of a variable, basically tricking the compiler into letting us force the static type of am expression. To make `largerPoodle` into a static type `Poodle` we will use the following:

```
Poodle largerPoodle = (Poodle) maxDog(frank, frankJr);
```

Note that we are not changing the actual dynamic type of maxDog- we are just telling the compiler what is coming out of maxDog will be a `Poodle` . This means that any reference to `largerPoodle` will have a static type of `Poodle` associated with it.

Casting, while powerful is also quite dangerous. You need to ensure that what you are casting to can and will actually happen. There are a few rules that can be used:

- You can always cast up (to a more generic version of a class) without fear of ruining anything because we know the more specific version is a version of the generic class. For example you can always cast a Poodle to a Dog because all Poodles are Dog's.
- You can also cast down (to a more specific version of a class) with caution as you need to make sure that, during runtime, nothing is passed in that violates your cast. For example, sometimes Dog's are Poodle's but not always.
- Finally, you cannot ever cast to a class that is above or below the class being cast. For an example, you cannot cast a Dog to a Monkey because a Monkey is not in the direct lineage of a Dog- it is a child of animal so a bit more distant.

Exercises

C Level

1. Do the problems from lecture
2. Is it possible for an interface to extend a class? Provide an argument as to why or why not.

3. What are the differences between inheritance through classes and interfaces? Is there a particular time when you would want to use one over the other?

B level

1. Say there is a class `Poodle` that inherits from `Dog`. The Dog class looks like this

   ```
   public class Dog{
       int weight;
       public Dog(int weight_in_pounds) {
         weight = weight_in_pounds;
       }
     }
   ```

   And the Poodle class looks like this.

   ```
   public class Poodle extends Dog{
     public Poodle() {}
   }
   ```

   Is this valid? If so explain why Poodle is a Dog if Dog has no constructor with no argument. If it is not valid then explain how we can make it valid.

2. The `Monkey` class is a subclass of the `Animal` class and the `Dog` class is a subclass of the `Animal` class. However a Dog is not a Monkey nor is a Monkey a Dog. What will happen for the following code? Assume that the constructors are all formatted properly.

   ```
   Monkey jimmy = new Monkey("Jimmy");
   Dog limmy = (Dog) jimmy;
   ```

3. How about for this code?

   ```
   Monkey orangutan = new Monkey("fruitful");
   Dog mangotan = (Dog)(Animal) orangutan;
   ```

   Provide brief explanation as to why you believe your answers to be correct.