Inheritance1 Study Guide | CS 61B Spring 2018

sp18.datastructur.es/materials/lectures/lec8/lec8

Lecture Code

Code from this lecture available at

https://github.com/Berkeley-CS61B/lectureCode-sp18/tree/master/Inheritance1.

Overview

Method Overloading In Java, methods in a class can have the same name, but different parameters. For example, a `Math` class can have an `add(int a, int b)` method and an `add(float a, float b)` method as well. The Java compiler is smart enough to choose the correct method depending on the parameters that you pass in. Methods with the same name but different parameters are said to be overloaded.

Making Code General Consider a `largestNumber` method that only takes an AList as a parameter. The drawback is that the logic for `largestNumber` is the same regardless of if we take an `AList` or `SLList`. We just operate on a different type of list. If we use our previous idea of method overriding, we result in a very long Java file with many similar methods. This code is hard to maintain; if we fix a bug in one method, we have to duplicate this fix manually to all the other methods.

The solution to the above problem is to define a new reference type that represents both `AList` and `SLList`. We will call it a `List`. Next, we specify an "is-a" relationship: An `AList` is a `List`. We do the same for `SLList`. Let's formalize this into code.

Interfaces We will use the keyword `interface` instead of `class` to create our `List`. More explicitly, we write:

```
public interface List<Item> { ... }
```

The key idea is that interfaces specify what this `List` can do, not how to do it. Since all lists have a `get` method, we add the following method signature to the interface class:

```
public Item get(int i);
```

Notice we did not define this method. We simply stated that this method should exist as long as we are working with a `List` interface.

Now, we want to specify that an `AList` is a `List`. We will change our class declaration of `AList` to:

```
public AList<Item> implements List<Item> { ... }
```

We can do the same for `SLList`. Now, going back to our `largestNumber` method, instead of creating one method for each type of list, we can simply create one method that takes in a `List`. As long as our actual object implements the `List` interface, then this method will work properly!

Overriding For each method in `AList` that we also defined in `List`, we will add an @Override right above the method signature. As an example:

```
@Override
public Item get(int i) { ... }
```

This is not necessary, but is good style and thus we will require it. Also, it allows us to check against typos. If we mistype our method name, the compiler will prevent our compilation if we have the @Override tag.

Interface Inheritance Formally, we say that subclasses inherit from the superclass. Interfaces contain all the method signatures, and each subclass must implement every single signature; think of it as a contract. In addition, relationships can span multiple generations. For example, C can inherit from B, which can inherit from A.

Default Methods Interfaces can have default methods. We define this via:

```
default public void method() { ... }
```

We can actually implement these methods inside the interface. Note that there are no instance variables to use, but we can freely use the methods that are defined in the interface, without worrying about the implementation. Default methods should work for any type of object that implements the interface! The subclasses do not have to re-implement the default method anywhere; they can simply call it for free. However, we can still override default methods, and re-define the method in our subclass.

Static vs. Dynamic Type Every variable in Java has a static type. This is the type specified when the variable is declared, and is checked at compile time. Every variable also has a dynamic type; this type is specified when the variable is instantiated, and is checked at runtime. As an example:

```
Thing a;
a = new Fox();
```

Here, `Thing` is the static type, and `Fox` is the dynamic type. This is fine because all foxes are things. We can also do:

```
Animal b = a;
```

This is fine, because all foxes are animals too. We can do:

```
Fox c = b;
```

This is fine, because `b` points to a `Fox`. Finally, we can do:

```
a = new Squid()
```

This is fine, because the static type of `a` is a `Thing`, and `Squid` is a thing.

Dynamic Method Selection The rule is, if we have a static type `X`, and a dynamic type `Y`, then if `Y` overrides the method from `X`, then on runtime, we use the method in `Y` instead. Student often confuse overloading and overriding.

Overloading and Dynamic Method Selection Dynamic method selection plays no role when it comes to overloaded methods. Consider the following piece of code, where `Fox extends Animal`.

```
1  Fox f = new Fox();
2  Animal a = f;
3  define(f);
4  define(a);
```

Let’s assume we have the following overloaded methods in the same class:

```
public static void define(Fox f) { ... }
public static void define(Animal a) { ... }
```

Line 3 will execute `define(Fox f)`, while line 4 will execute `define(Animal a)`. Dynamic method selection only applies when we have overridden methods. There is no overriding here, and therefore dynamic method selection does not apply.

Exercises