Inheritance3 Study Guide | CS 61B Spring 2018

sp18.datastructur.es/materials/lectures/lec10/lec10

Lecture Code

Code from this lecture available at

https://github.com/Berkeley-CS61B/lectureCode-sp18/tree/master/inheritance3.

Overview

Review: Typing Rules

- Compiler allows the memory box to hold any subtype.
- Compiler allows calls based on static type.
- Overriden non-static methods are selected at runtime based on dynamic type.
- For overloaded methods, the method is selected at compile time.

Subtype Polymorphism Consider a variable of static type `Deque`. The behavior of calling `deque.method()` depends on the dynamic type. Thus, we could have many subclasses the implement the `Deque` interface, all of which will be able to call `deque.method()`.

Subtype Polymorphism Example Suppose we want to write a function `max()` that returns the max of any array regardless of type. If we write a method `max(Object[] items)`, where we use the '>' operator to compare each element in the array, this will not work! Why is this the case?

Well, this makes the assumption that all objects can be compared. But some objects cannot! Alternatively, we could write a `max()` function inside the Dog class, but now we have to write a `max()` function for each class that we want to compare! Remember, our goal is to write a "one true max method" that works for all comparable objects.

Solution: OurComparable Interface The solution is to create an interface that contains a `compareTo(Object)` method; let's call this interface `OurComparable`. Now, our `max()` method can take a `OurComparable[]` parameter, and since we guarantee that any object which extends the interface has all the methods inside the interface, we guarantee that we will always be able to call a `compareTo` method, and that this method will correctly return some ordering of the objects.

Now, we can specify a "one true max method". Of course, any object that needs to be compared must implement the `compareTo` method. However, instead of re-implementing the `max` logic in every class, we only need to implement the logic for picking the ordering of the objects, given two objects.

Even Better: Java's In-Built Comparable Java has an in-built `Comparable` interface that uses generics to avoid any weird casting issues. Plus, Comparable already works for things like `Integer`, `Character`, and `String`; moreover, these objects have already implemented a `max`, `min`, etc. method for you. Thus you do not need to re-do work that's already been done!

Comparators The term "Natural Order" is used to refer to the ordering implied by a `Comparable`'s `compareTo` method. However, what if we want to order our `Dog` objects by something other than `size`? We will instead pass in a `Comparator<T>` interface, which demands a `compare()` method. We can then implement the `compare()` method anyway we want to achieve our ordering.

Exercises