Lecture 10: Subtype Polymorphism vs. HoFs

9/18/2020

Static Methods, Variables, and Inheritance

- You may find questions on old 61B exams, worksheets, etc. that consider:
  - What if a subclass has variables with the same name as a superclass?
  - What if subclass has a static method with the same signature as a superclass method?
    - For static methods, we do not use the term overriding for this
- These two practices above are called "hiding"
  - Bad style
  - There is no good reason to ever do this
  - The rules for resolving the conflict are a bit confusing to learn
  - Will not be taught or tested in 61B

Subtype Polymorphism

Subtype Polymorphism

- The biggest idea of the last couple of lectures: **subtype polymorphism**
  - Polymorphism: "providing a single interface to entities of different types"
- Consider a variable `deque` of static type `Deque`:
  - When you call `deque.addFirst()`, the actual behavior is based on the dynamic type
  - Java automatically selects the right behavior using what is sometimes called "dynamic method selection"

Subtype Polymorphism vs. Explicit Higher Order Functions

- Suppose we want to write a program that prints a string representation of the larger of two objects
- Explicit HoF Approach

```
def print_larger(x, y, compare, stringify):
    if compare(x, y):
        return stringify(x)
    return stringify(x)
```

- Subtype Polymorphism Approach

```
def print_larger(x, y):
    if x.largerThan(y):
        return x.str()
    return y.str()
```

DIY Comparison

shoutkey.com/TBA

- Suppose we want to write a function `max()` that returns the max of any array, regardless of type

Writing a General Max Function: The Fundamental Problem

- Objects cannot be compared to objects with `>`
  - One (bad) way to fix this: Write a max method in the Dog class
  - What is the disadvantage of this?
    - Given up dream of one true `max` function
    - You will need a `max` function for each new class
    - Redundant code
  - How could we fix our Maximizer class using inheritance / HoFs?

Solution

- Create an interface that guarantees a comparison method
  - Have Dog implement this interface
  - Write Maximizer class in terms of this interface

```
public static OurComparable max(OurComparable[] items) {...}
```

```
public interface OurComparable {
    /** Return negative number if this is less than o
     * Return 0 if this is equal to o
     * Return positive number if this if greater than o
     */
    public int compareTo(Object obj);
}

public class Dog implements OurComparable {
    private String name;
    private int size;

    public Dog(String n, int s) {
        name = n;
        size = s;
    }

    public void bark() {
        System.out.println(name + " says: bark");
    }

    // Returns negative number if this dog is less than the dog pointed by
o, and so forth
    public int compareTo(Object o) {
        Dog otherDog = (Dog) o;
```

```
        return this.size - otherDog.size;
    }
}

public class Maximizer {
    public static OurComparable max(OurComparable[] items) {
        int maxDex = 0;
        for (int i = 0; i < items.length; i += 1) {
            int cmp = items[i].compareTo(items[maxDex])
            if (cmp > 0) {
                maxDex = i;
            }
        }
        return items[maxDex];
    }
}
```

The OurComparable Interface

- Specification, returns:
  - Negative number if **this** is less than obj
  - 0 if **this** is equal to object
  - Positive number if **this** is greater than obj

General Maximization Function Through Inheritance

- Benefits of this approach
  - No need for array maximization code in every custom type (i.e. no Dog.maxDog(Dog[]) function required)
  - Code that operates on multiple types (mostly) gracefully

Comparables

The Issues with OurComparable

- Two issues:
  - Awkward casting to/from Objects
  - We made it up
    - No existing classes implement OurComparable (e.g. string, etc)
    - No existing classes use OurComparable (e.g. no built-in max function that uses OurComparable)
- The industrial strength approach: Use the built-in Comparable interface
  - Already defined nad used by tons of libraries. Uses generics.

```
public class Dog implements Comparable<Dog> {

    ...

    // Returns negative number if this dog is less than the dog pointed by
```

```
o, and so forth
    public int compareTo(Dog otherDog) {
        return this.size - otherDog.size;
    }
}

public class Maximizer {
    public static Comparable max(Comparable[] items) {
        int maxDex = 0;
        for (int i = 0; i < items.length; i += 1) {
            int cmp = items[i].compareTo(items[maxDex])
            if (cmp > 0) {
                maxDex = i;
            }
        }
        return items[maxDex];
    }
}
```

Comparable Advantages

- Lots of built in classes implement Comparable (e.g. String)
- Lots of libraries use the Comparable interface (e.g. Arrays.sort)
- Avoids need for casts

Comparators

Natural Order

- The term "Natural Order" is used to refer to the ordering implied by a `Comparable`'s `compareTo` method
  - Example: Dog objects (as we've defined them) have a natural order given by their size
- May wish to order objects in a different way
  - Example: by name

Additional Orders in Java

- The standard Java approach: Create `sizeComparator` and `nameComparator` classes that implement the `Comparator` interface
  - Requires methods that also take Comparator arguments

```
import java.util.Comparator;

public class Dog implements OurComparable {

    ...

    // Returns negative number if this dog is less than the dog pointed by
o, and so forth
    public int compareTo(Object o) {
```

```
        Dog otherDog = (Dog) o;
        return this.size - otherDog.size;
    }

    private static class NameComparator implements Comparator<Dog> {
        public int compare(Dog a, Dog b) {
            return a.name.compareTo(b.name);
        }
    }

    public static Comparator<Dog> getNameComparator() {
        return new NameComparator();
    }
}


import java.util.Comparator;

public class DogLauncher {
    public static void main(String[] args) {

        ...

        Comparator<Dog> nc = new Dog.getNameComparator();
        if (nc.compare(d1, d3) > 0) {
            d1.bark();
        } else {
            d2.bark();
        }
    }
}
```

Comparable and Comparator Summary

- Interfaces provide us with the ability to make **callbacks**
  - Sometimes a function needs the help of another function that might not have been written yet
    - Example: `max` needs `compareTo`
    - The helping function is sometimes called a "callback"
  - Some languages handle this using explicit function passing
  - In Java, we do this by wrapping up the needed function in an interface (e.g. `Arrays.sort` needs `compare` which lives inside the `comparator` interface)
  - `Arrays.sort` "call back" whenever it needs a comparison
    - Similar to giving your number to someone if they need information