Lecture 8: Inheritance, Implements

9/14/2020

The Desire for Generality

AList and SList

- After adding the insert methods from discussion 3, our AList and SLList classes have the following methods (exact same method signatures for both classes)

Using ALists and SLists: WordUtils.java

- Suppose we're writing a library to manipulate lists of words. Might want to write a function that finds the longest word from a list
- Suppose we also want to be able to handle ALists. What should we change?
- What if we want to be able to handle both?

Method Overloading in Java

- Java allows multiple methods with the same name, but with different parameters
    - This is called method **overloading**

```
public static String longest(AList<String> list) {
    ...
}
public static String longest(SLList<String> list) {
    ...
}
```

The Downsides

- While overloading works, it is a bad idea in the case of `longest`
    - Source code files are unnecessary long
    - Repeating yourself is aesthetically gross
    - **More code to maintain**
        - Any change made to one, must be made to another
        - Including bug fixes!
    - suppose we make another list someday, we'll need yet another function

Hypernyms, Hyponyms, and Interface Inheritance

Hypernyms

- Washing your poodle:
    - Brush your poodle before a bath

    - Use lukewarm water
    - Talk to your poodle in a calm voice
    - Use poodle shampoo
    - Rinse well
    - Air-dry
    - Reward your poodle
- Washing your malamute
    - Brush your malamute before a bath
    - Use lukewarm water
    - Talk to your malamute in a calm voice
    - Use malamute shampoo
    - Rinse well
    - Air-dry
    - Reward your malamute
- In natural languages (e.g. English), we have a concept known as "hypernym" to deal with this problem
    - Dog is a "hypernym" of poodle, malamute, yorkie, etc.

Hypernym and Hyponym

- We use the word hyponym for the opposite type of relationship
    - "dog": Hypernym of "poodle", "malamute"
    - "poodle": Hyponym of "dog"
- Hypernyms and hyponyms compose a hierarchy
    - A dog "is-a" canine
    - A canine "is-a" carnivore

Simple Hyponymic Relationships in Java

- SLLists and ALists are both clearly some kind of "list"
    - List is a hypernym of SLList and AList
- Expressing this in Java is a two-step process:
    - Define a reference type for our hypernym (List61B.java)
    - Specify that SLLists and ALists are hyponyms of that type

Step 1: Defining a List61B.java

- We'll use the new keyword **interface** instead of **class** to define a List61B
    - Idea: Interface is a specification of what a List is able to do, not how to do it

```
public interface List61B<Item> {
    public void addLast(Item x);
    public Item getLast();
    public Item get(int i);
    public int size();
    public Item removeLast();
    public void insert(Item x, int position);
    public Item getFirst();
}
```

Step 2: Implementing the List61B Interface

- We'll now:
  - Use the new **implements** keyword to tell the Java compiler that SLList and AList are hyponyms of List61B

```
public class AList<Item> implements List61B<Item> {
    ...
}

public class SLList<Item> implements List61B<Item> {
    ...
}
```

```
public class WordUtils {
    public static String longest(List61B<String> list) {
        ...
    }
}
```

Overriding vs. Overloading

Method Overriding

- If a "subclass" has a method with the exact same signature as in the "superclass", we say the subclass **overrides** the method
  - e.g. AList **overrides** addLast(Item)
  - Methods with the same name but different signatures are **overloaded**

```
public class Math {
    public int abs(int a)
    public double abs(double a)
}
```

- abs is **overloaded**

Optional Step 2B: Adding the @Override Annotation

- In 61B, we'll always mark every overriding method with the `@Override` annotation
  - Example: Mark AList.java's overriding methods with `@Override`
  - The only effect of this tag is that the code won't compile if it is not actually an overriding method
- Why use `@Override`?
  - Main reason: Protects against typos

        - If you say `@Override`, but the method isn't actually overriding anything, you'll get a compile error
        - e.g. `public void addLast(Item x)`
    - Reminds programmer that method definition came from somewhere higher up in the inheritance hierarchy e.g.

```
public class AList<Item> implements List61B<Item> {
    @Override
    public Item getItem(int a) {
        ...
    }
}
```

Interface Inheritance

Interface Inheritance

- Specifying the capabilities of a subclass using the **implements** keyword is known as **interface inheritance**
    - Interface: The list of al method signatures
    - Inheritance: The subclass "inherits" the interface from a superclass
    - Specifies what the subclass can do, but not how
    - Subclasses **must** override all of these methods!
        - Will fail to compile otherwise
    - Such relationships can be multi-generational
        - Figure: Interfaces in white, classes in green
- Interface inheritance is a powerful tool for generalizing code
    - WordUtils.longest works on SLLists, ALists, and even lists that have not yet been invented

Copying the Bits

- Two seemingly contradictory facts:
    - #1: When you set `x = y` or pass a parameter, you're just copying the bits
    - #2: A memory box can only hold 64 bit addresses for the appropriate type
- Answer: If X is a superclass of , then memory boxes for X may contain Y
    - An `AList` is-a List
    - Therefore `List` variables can hold `ALList` addresses
- e.g. the following works just fine:

```
public static void main(String[] args) {
    List61B<String> someList = new SLList<>();
    someList.addFirst("elk");
}
```

Implementation Inheritance: Default Methods

Implementation Inheritance

- Interface Inheritance:
  - Subclass inherits signatures, but NOT implementation
- Java also allows **implementation inheritance**
  - Subclasses can inherit signatures AND implementation
- Use the **default** keyword to specify a method that subclasses should inherit from an **interface**
  - Ex. add a default `print()` to List61B

```
public interface List61B<Item> {
    public void addLast(Item x);
    public Item getLast();
    public Item get(int i);
    public int size();
    public Item removeLast();
    public void insert(Item x, int position);
    public Item getFirst();

    /** Prints out the entire List. */
    default public void print() {
        for (int i = 0; i < size(); i += 1) {
            System.out.print(get(i) + ' ');
        }
        System.out.println();
    }
}
```

Is the print() method efficient?

- print() is efficient for AList and inefficient for SLList
  - See the `get` method for both classes

Overriding Default Methods

Overriding Default Methods

- If you don't like the default method, you can override it
  - Any call to `print()` on an SList will use this method instead of default
  - Use `@Override` to cate typos like `public void pirnt()`

```
public class SLList<Item> {
    @Override
    public void print() {
        for (Node p = sentinel.next; p != null; p = p.next) {
            System.out.print(p.item + ' ');
        }
    }
}
```

Dynamic Method Selection

Static Type vs. Dynamic Type

- Every variable in Java has a "compile-time type", aka "static type"
  - This is the type specified at **declaration**. Never changes!
- Variables also have a "run-time type", aka "dynamic type"
  - This is the type specified at **instantiation** (e.g. when using `new`)
  - Equal to the type of the object being pointed at

Dynamic Method Selection For Overridden Methods

- Suppose we call a method of an object using a variable with:
  - compile-time type X
  - run-time type Y
- Then if Y **overrides** the method, Y's method is used instead
  - This is known as "dynamic method selection"

More Dynamic Method Selection, Overloading vs. Overriding

The Method Selection Algorithm

- Consider the function called `foo.bar(x1)` where `foo` has static type `TPrime`, and `x1` has static type `T1`
- At compile time, the compiler verifies that `TPrime` has a method that can handle `T1`. It then records the signature of this method
  - Note: If there are multiple methods that can handle `T1`, the compiler records the "most specific" one.
- At runtime, if `foo`'s dynamic type overrides the **recorded signature**, use the overridden method. Otherwise, use `TPrime`'s version of the method

Is a vs Has a, Interface vs Implementation Inheritances

Interface vs. Implementation Inheritance

- Interface inheritance (aka what):
  - Allows you to generalize code in a powerful, simple way
- Implementation Inheritance (aka how):
  - Allows code-reuse: Subclasses can rely on superclasses or interfaces
    - Example: `print()` implemented in List61B.java
    - Gives another dimension of control to subclass designers: Can decide whether or not to override default implementations
- Important: In both cases, we specify "is-a" relationships, not "has-a"
  - Good: Dog implements Animal, SLList implements List61B
  - Bad: Cat implements Claw, Set implements SLList

Dangers of Implementation Inheritance

- Particular dangers of implementation inheritance
    - makes it harder to keep track of where something was actually implemented
    - Rules for resolving conflicts can be arcane
        - Ex: What if two interfaces both give conflicting default methods?
    - Encourages overly complex code
        - Common mistake: Has-a vs Is-a!
    - Breaks encapsulation!