Lecture 9: Extends, Casting, Higher Order Functions

9/16/2020

Implementation Inheritance: Extends

The Extends Keyword

- When a class is a hyponym of an interface, we use **implements**
- If you want one class to be a hyponym of another *class* (instead of interface), you use **extends**
- We'd like to build a RotatingSLList that can perform any SLList operation as well as:
  - `rotateRight()`: Moves back item to the front

RotatingSLList

- Because of **extends**, RotatingSLList inherits all members of SLList:
  - All instance and static variables
  - All methods
  - All nested classes
- Constructors are not inherited

```
public class RotatingSLList<Item> extends SLList<Item> {
    // Rotates list to the right
    public void rotateRight() {
        Item x = removeLast();
        addFirst(x);
    }
}
```

Another Example: VengefulSLList

- Suppose we want to build an SLList that:
  - Remembers all items that have been destroyed by `removeLast`
  - Has an additional method `printLostItems()`, which prints all deleted items

```
public class VengefulSLList<Item> extends SLList<Item> {
    SLList<Item> deletedItems;

    public VengefulSLList() {
        super();  // Optional line
        deletedItems = new SLList<Item>();
    }

    @Override
    public Item removeLast() {
        Item x = super.removeLast();  // Calls Superclass's version of
removeLast()
        deletedItems.addLast(x);
        return x;
    }

    // Prints deleted items
    public void printLostItems() {
        deletedItems.print();
    }
}
```

Constructor Behavior is Slightly Weird

- Constructors are not inherited. However, the rules of Java say that **all constructors must start with a call to one of the super class's constructor**
  - Idea: If every VengefulSLList is-an SLList, every VengefulSLList must be set up like an SLList
    - If you didn't call SLList constructor, sentinel would be null. Very bad.
  - You can explicitly call the constructor with the keyword super (no dot)
  - If you do not explicitly call the constructor, Java will automatically do so for you

Calling Other Constructors

- If you want to use a super constructor other than the no-argument constructor, can give parameters to super

```
public class VengefulSLList<Item> extends SLList<Item> {
    SLList<Item> deletedItems;

    public VengefulSLList() {
        super();  // Optional line
        deletedItems = new SLList<Item>();
    }

    public VengefulSLList(Item x) {
        super(x);  // NOT OPTIONAL! (calls no-argument constructor
otherwise)
        deletedItems = new SLList<Item>();
    }
}
```

The Object Class

- As it happens, every type in Java is a descendant of the Object class
  - VengefulSLList extends SLList
  - SLList extends Object (implicitly)
- Interfaces do not extend the object class

Is-a vs. Has-A

- Important Note: extends should only be used for **is-a** (hypernymic) relationships
- Common mistake is to use it for "**has-a**" relationships

Encapsulation

Complexity: The Enemy

- When building large programs, our enemy is complexity
- Some tools for managing complexity
    - Hierarchical abstraction
        - Create **layers of abstraction**, with clear abstraction barriers
    - "Design for change" (D. Parnas)
        - Organize program around objects
        - Let objects decide how things are done
        - **Hide information** others don't need
- Managing complexity supremely important for large projects (e.g. project 2)

Modules and Encapsulation

- **Module**: A set of methods that work together as a whole to perform some task or set of related tasks
- A module is said to be **encapsulated** if its implementation is *completely hidden*, and it can be accessed only through a documented interface
    - Instance variable private. Methods like `resize` private

A Cautionary Tale

- Interesting questions from project 1B
    - How can we check the length of StudentArrayDeque?
    - Private access in given classes
    - Can we assume these things about StudentArrayDeque?

Abstraction Barriers

- As the user of an ArrayDeque, you cannot observe its internals
    - Even when writing tests, you don't (usually) want to peer inside
- Java is a great language for enforcing abstraction barriers with syntax

Implementation Inheritance Breaks Encapsulation

- What would vd.barkMany(3) output? (vd is a VerboseDog)
    - An infinite loop!

```
public void bark() {
    barkMany(1);
}
public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        System.out.println("bark");
    }
}

@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ") {
        for (int i = 0; i < N; i += 1) {
            bark();  // calls inherited bark method
        }
    }
}
```

Type Checking and Casting

Reminder: Dynamic Method Selection

- If overridden, decide which method to call based on **run-time** type (dynamic type) of variable

Compile-Time Type Checking

- Compiler allows method calls based on **compile-time** type (static type) of variable
- Compiler also allows assignments based on compile-time types
    - Compiler plays it as safe as possible with type checking

Compile-Time Types and Expressions

- Expressions have compile-time types
    - An expression using the new keyword has the specified compile-time type
- `SLList<Integer> s1 = new VengefulSLList<Integer>();`
    - Compile-time type of right hand side (RHS) expression is VengefulSLList
    - A VengefulSLList is-an SLList, so assignment is allowed
- `VengefulSLList<Integer> vs1 = new SLList<Integer>();`
    - Compile-time type of RHS expression is SLList
    - An SLList is not necessarily a VengefulSLList, so compilation error results

Compile-Type Types and Expressions

- Expressions have compile-time types
    - Method class have compile-time type equal to their declared type
- `public static Dog maxDog(Dog d1, Dog d2) {...}`
    - Any call to maxDog will have compile-time type Dog!
- Example:

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

dog largerDog = maxDog(frank, frankJr);
Poodle largerPoodle = maxDog(frank, frankJr);  // Compilation Error!
// RHS has compile-time type Dog
```

Casting

- Java has a special syntax for forcing the compile-time type of any expression
  - Put desired type in parenthesis before expression
  - Examples:
    - Compile-time type Dog: `maxDog(frank, frankJr);`
    - Compile-time type Poodle: `(Poodle) maxDog(frank, frankJr);`
- Think of it as a way to trick the compiler

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

dog largerDog = maxDog(frank, frankJr);
Poodle largerPoodle = (Poodle) maxDog(frank, frankJr);  // Compilation OK!
// RHS has compile-time type Poodle
```

- Casting is a powerful but dangerous tool
  - Tells Java to treat an expression as having a different compile-time type
  - Effectively tells the compiler to ignore its type checking duties

```
Poodle frank = new Poodle("Frank", 5);
Malamute frankSr = new Malamute("Frank Sr.", 100);

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

- If we run the code above, we get a ClassCastException at runtime

Higher Order Functions (A First Look)

Higher Order Functions

- Higher Order Function: A function that treats another function as data
  - e.g takes a function as input
- Example in Python:

```
def tenX(x):
    return 10*x

def do_twice(f, x):
    return f(f(x))

print(do_twice(tenX, 2))
```

Higher Order Functions in Java 7

- Old School (Java 7 and earlier)
    - Fundamental issue: Memory boxes (variables) cannot contain pointers to functions
- Can use an interface instead. Let's try it out

```
// Represents a function that takes in an integer, and returns an integer
public interface IntUnaryFunction {
    int apply(int x);
}
```

```
public class TenX implements IntUnaryFunction {
    /** Returns ten times the argument */
    public int apply(int x) {
        return 10 * x;
    }
}
```

```
// Demonstrates higher order functions in Java
public class HofDemo {
    public static int doTwice(IntUnaryFunction f, int x) {
        return f.apply(f.apply(x));
    }

    public static void main(String[] args) {
        IntUnaryFunction tenX = new TenX();
        System.out.println(doTwice(tenX, 2));
    }
}
```

- Very verbose

Implementation Inheritance Cheatsheet

- VengefulSLList extends SLList means a VengefulSLList is-an SLList. Inherits all members!
    - Variables, methods, nested classes
    - Not constructors
    - Subclass constructor must invoke superclass constructor first
    - Use super to invoke overridden superclass methods and constructors
- Invocation of overridden methods follows two simple rules:
    - Compiler plays it safe and only lets us do things allowed by **static** type
    - For overridden methods the actual method invoked is based on **dynamic** type of invoking expressions
        - Does not apply to **overloaded** methods!
    - Can use casting to overrule compiler type checking.