**4.1**

Exercise 4.1.1. Suppose we want to write a class WordUtilsthat includes functions we can run on lists of words, including a method that calculates the longest string in an SLList. Try writing this method by yourself. The method should take in an SLList of strings and return the longest string in the list.

public static String longest(SLList list) {

        int maxDex = 0;

        for (int i = 0; i < list.size(); i += 1) {

            String longestString = list.get(maxDex);

            String thisString = list.get(i);

            if (thisString.length() > longestString.length()) {

                maxDex = i;

            }

        }

        return list.get(maxDex);

}

**Hypernyms, Hyponyms, and Interface Inheritance**

上义词，下义词，接口继承

public class AList<Item> implements List61B<Item> {

// Implements 是关键字

}

implements List61B<Item> 实际上是一种承诺。AList 类声明：“我承诺我将拥有并定义 List61B 接口中指定的所有属性和行为。”

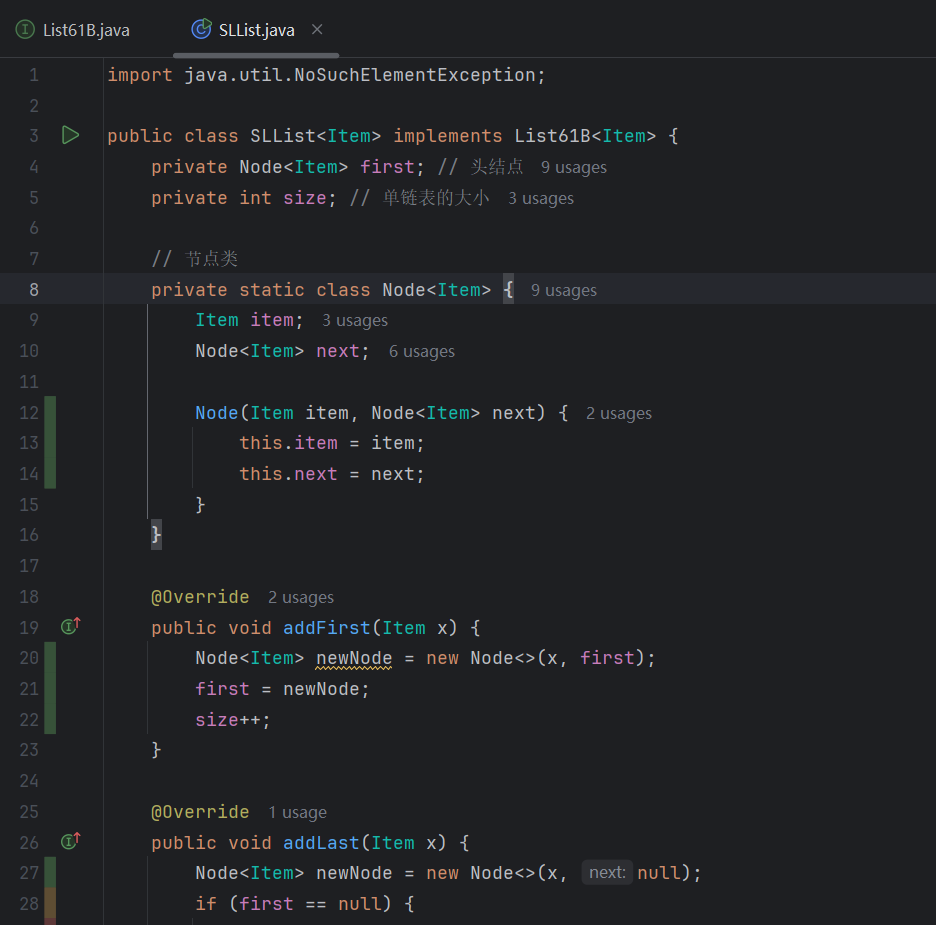
然后：

@Override要写在方法的上面

也就是说，我在父类里有个函数，叫sizeOf，然后我在子类里，我写了@Override，我仍然用sizeOf的函数名，但是我可以更改函数的内容  
**方法名：** 子类方法的方法名必须与父类方法相同，即 sizeOf。  
**参数列表：** 子类方法的参数列表必须与父类方法相同。  
**返回类型：** 子类方法的返回类型必须与父类方法相同，或者是其子类型（对于协变返回类型的情况）

实例已经放在Week4文件夹，父类和子类需要在同一目录、两个文件；@Override不需要引用任何包。





List61B<String> lst = new SLList<String>()

**静态类型**：指的是变量声明时的类型。例如，List61B<String> lst 中的 lst 的静态类型是 List61B<String>。

**动态类型**：指的是变量实际引用的对象的类型。例如，SLList<String> SP = new SLList<String>(); 中的 SP 的动态类型是 SLList<String>，因为它实际上指向了一个 SLList 的实例。

在非重载的情况下，List61B<String> lst = new SLList<String>(); 中的 lst 在编译时静态类型是 List61B<String>，但在运行时动态类型是 SLList<String>。这意味着，尽管你使用了 List61B 类型声明了 lst，但实际上 lst 指向的对象是 SLList 类型的。因此，你可以调用 lst 引用对象上 List61B 接口定义的方法，但实际运行时会调用 SLList 中的具体实现。

方法重载和动态分派：  
方法重载是指在同一个类中可以有多个方法，它们具有相同的名称但是参数类型或数量不同。Java 在调用重载方法时是根据静态类型来决定调用哪个方法的，这被称为静态分派。例如，考虑下面的两个方法：

public static void peek(List61B<String> list) {

System.out.println(list.getLast());

}

public static void peek(SLList<String> list) {

System.out.println(list.getFirst());

}

在调用 peek(SP); 和 peek(LP); 时，编译器会根据静态类型来选择调用哪个方法。因为 SP 的静态类型是 SLList<String>，所以会调用接受 SLList<String> 参数的 peek 方法。而 LP 的静态类型是 List61B<String>，会调用接受 List61B<String> 参数的 peek 方法。

**4.2**

**Extends**

**继承的定义**：你可以使用 extends 关键字来设置类之间的继承关系。例如，对于 RotatingSLList 类继承自 SLList 类，你的类头部会像这样：

public class RotatingSLList<Item> extends SLList<Item>{

}

这里，RotatingSLList<Item> 表示 RotatingSLList 是 SLList 的子类，并且拥有 SLList 类的所有属性和方法，同时可以在其基础上进行修改和扩展。

通过继承，你可以保留 SLList 的所有功能，包括 addFirst、size 等方法，并且可以在 RotatingSLList 中添加新的方法或修改现有方法来满足新的需求

例：

public void rotateRight() {

Item x = removeLast();

addFirst(x);

}

使用 extends 关键字，子类会继承父类的所有成员。"成员" 包括：

所有实例变量和静态变量  
所有方法  
所有嵌套类  
\*需要注意的是，构造函数不会被继承，而且私有成员不能被子类直接访问。

在Java中，当我们创建一个子类的对象时，如果没有显式调用父类的构造函数（使用 super()），Java会自动调用父类的无参构造函数。这确保了父类的必要初始化工作会被完成。在上述代码中，VengefulSLList 类的构造函数通过 super() 调用了父类 SLList 的无参构造函数，确保了在初始化 VengefulSLList 对象时，父类 SLList 的相关初始化工作会被执行。

虽然不调用 super() 也能正常工作（因为Java会隐式调用父类的无参构造函数），但显式地调用 super() 使代码更清晰，显示了父类构造函数的调用顺序和初始化过程，有助于代码的可读性和维护性。

上面是无参情况；

有参：

public VengefulSLList(Item x) {

super(x);

deletedItems = new SLList<Item>();

}

Encapsulation 封装

Using the private keyword in Java, it becomes virtually impossible to look inside an object - ensuring that the underlying complexity isn't exposed to the outside world.

例：

public class Dog {

public void bark() {

barkMany(1);

}

public void barkMany(int N) {

for (int i = 0; i < N; i += 1) {

System.out.println("bark");

}

}

}

public class VerboseDog extends Dog {

@Override

public void barkMany(int N) {

System.out.println("As a dog, I say: ");

for (int i = 0; i < N; i += 1) {

bark(); // 注意这里调用的是父类的 bark() 方法

}

}

public static void main(String[] args) {

VerboseDog vd = new VerboseDog();

vd.barkMany(3);

}

}

//重复无限次

**Casting 类型转换**类型转换是一个强大但危险的工具。实质上，类型转换告诉编译器不要执行其类型检查的职责 - 它要求你相信并按照你的意愿执行。以下是可能出现的一个问题：

Poodle frank = new Poodle("Frank", 5);

Malamute frankSr = new Malamute("Frank Sr.", 100);

Poodle largerPoodle = (Poodle) maxDog(frank, frankSr); // 运行时异常！

在这个例子中，我们比较了一个 Poodle 和一个 Malamute。如果没有进行类型转换，编译器通常不会允许调用 maxDog，因为右侧的编译时类型应该是 Dog，而不是 Poodle。然而，类型转换允许这段代码通过编译。当 maxDog 在运行时返回 Malamute 时，我们尝试将一个 Malamute 强制转换为 Poodle，这时会引发运行时异常 - ClassCastException（类转换异常）。  
因此，类型转换是一种强大的机制，但需要谨慎使用，因为它绕过了编译器的类型检查，可能导致运行时出现异常。

**高阶函数**

我们编写一个接口，定义任何接受整数并返回整数的函数 - IntUnaryFunction。

public interface IntUnaryFunction {

int apply(int x);

}

现在，我们可以编写一个实现 IntUnaryFunction 接口的类来表示具体的函数。我们创建一个函数，接受一个整数并返回这个整数的10倍。

public class TenX implements IntUnaryFunction {

public int apply(int x) {

return 10 \* x;

}

}

现在让我们来编写 do\_twice 函数。

public static int do\_twice(IntUnaryFunction f, int x) {

return f.apply(f.apply(x));

}

在Java中调用 do\_twice(tenX, 2) 的打印语句如下所示：

System.out.println(do\_twice(new TenX(), 2));

这段代码的输出将是 200。

**4.3**

显式高阶函数方法提供了一种通用的、不依赖于具体对象类型的解决方案，而子类型多态方法则允许对象根据其具体类型来做出选择。

In java, Objects cannot be compared with >

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");

}

public int compareTo(Object o) {

Dog uddaDog = (Dog) o;

if (this.size < uddaDog.size) {

return -1;

} else if (this.size == uddaDog.size) {

return 0;

}

return 1;

}

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];

}

}

Notice that since compareTo takes in any arbitrary Object o, we have to cast the input to a Dog to make our comparison using the size instance variable.

**4.4**

**Exercise 4.4.1** Write a method getWords that takes in a String inputFileName and puts every word from the input file into a list. Recall how we read words from a file in proj0. \*Hint: use In

**Exercise 4.4.2** Write a method countUniqueWords that takes in a List<String> and counts how many **unique** words there are in the file

public static List<String> getWords(String inputFileName) {

List<String> lst = new ArrayList<String>();

In in = new In();

while (!in.isEmpty()) {

lst.add(in.readString()); //optionally, define a cleanString() method that cleans the string first.

}

return lst;

}

public static int countUniqueWords(List<String> words) {

Set<String> ss = new HashSet<>();

for (String s : words) {

ss.add(s);

}

return ss.size();

}