Getting Started

[Inheritance2, Video 1] Basic Use of Extends

The Extends Keyword

Up to now we have been writing classes and interfaces, and you may have noticed places where we have to write redundant code for different or similar classes. So, we have the idea of **inheritance**: the idea that a class/object does not need to redefine all its methods, and instead can use properties of a parent class.

When a class is a hyponym of an interface, we used implements. **Example below:**

```
SLList<Blorp> implements List61B<Blorp>
```

If you want one class to be a hyponym of another class (instead of an interface), you use extends.

Rotating SLList

We'd like to build RotatingSLList that can perform any SLList operation as well as: rotateRight(): Moves back item to the front.

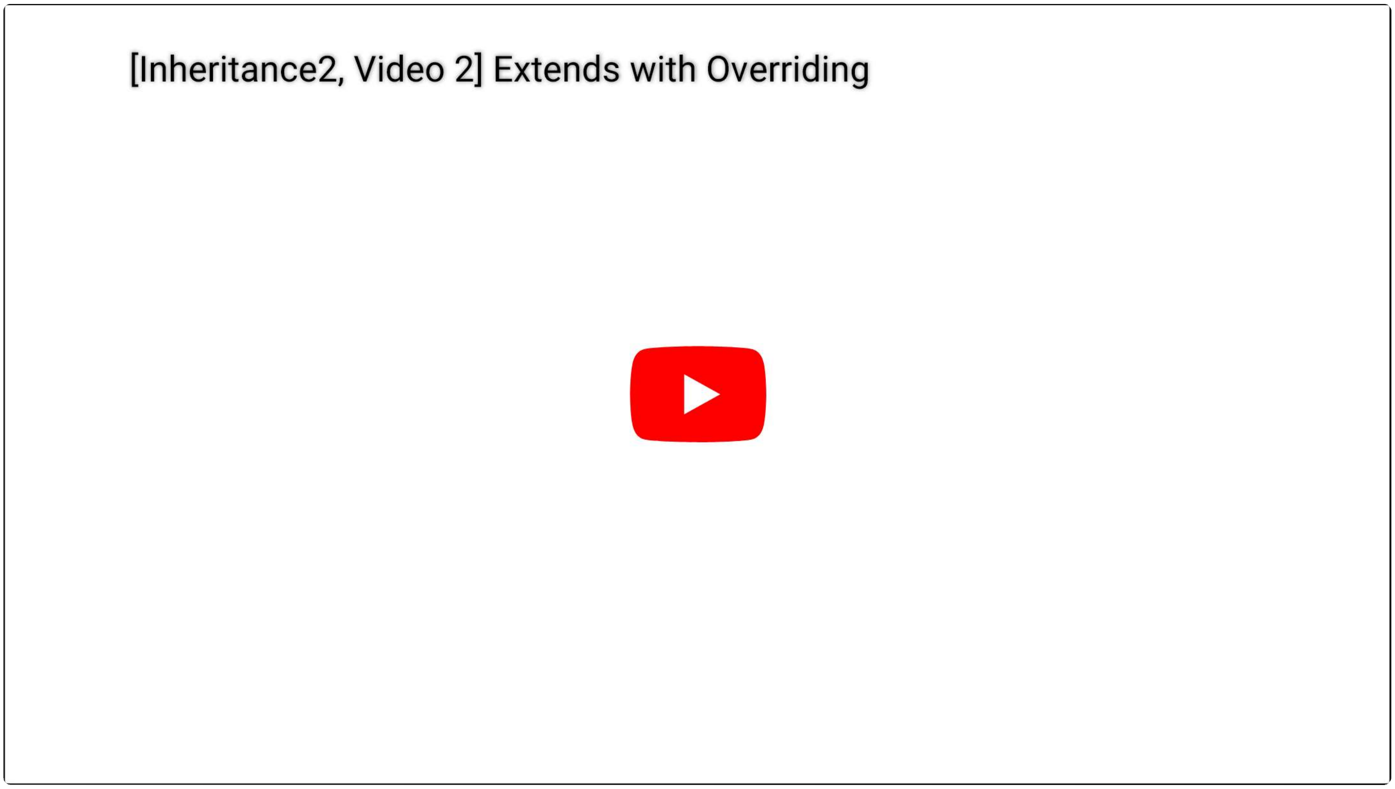
```
public class RotatingSLList<Blorp> extends SLList<Blorp>{
       public void rotateRight() {
              Blorp oldBack = removeLast();
              addFirst(oldBack);
    }
}
```

Because of extends, RotatingSLList inherits all members of SLList:

- All instance and static variables.
- All methods.
- All nested classes.
- Constructors are **not** inherited!

Example: Suppose we have [5, 9, 15, 22]. After rotateRight: [22, 5, 9, 15].

Video Example



[Inheritance2, Video 2] Extends with Overriding

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> {
    private SLList<Item> deletedItems;
    public VengefulSLList() {
       deletedItems = new SLList<Item>();
    }

    @Override
    public Item removeLast() {
            Item oldBack = super.removeLast(); /*calls Superclass’s
version of removeLast() */
            deletedItems.addLast(oldBack);
            return oldBack;
    }

    public void printLostItems() {
            deletedItems.print();
    }
}

public static void main(String[] args) {
        VengefulSLList<Integer> vs1 = new VengefulSLList<Integer>();
        vs1.addLast(1);
        vs1.addLast(5);
        vs1.addLast(10);
        vs1.addLast(13);      /* [1, 5, 10, 13] */
        vs1.removeLast();     /* 13 gets deleted. */
        vs1.removeLast();     /* 10 gets deleted. */
        System.out.print("The fallen are: ");
        vs1.printLostItems(); /* Should print 10 and 13. */
}
```

Constructor Behavior

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 constructors [Link].

- 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 don’t explicitly call the constructor, Java will automatically do it for you.

These constructors below are exactly equivalent:

```
public VengefulSLList() {
   deletedItems = new SLList<Item>();
}

public VengefulSLList() {
   super();
   deletedItems = new SLList<Item>();
}
```

On the other hand, if you want to use a super constructor other than the no-argument constructor, can give parameters to super.

These constructors below are **not** equivalent! The code to below makes implicit call to super(), not super(x). This is because only the empty argument super() is called.

```
public VengefulSLList(Item x) {
   super(x);
   deletedItems = new SLList<Item>();
}

public VengefulSLList(Item x) {
   deletedItems = new SLList<Item>();
}
```

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. (a.k.a. meronymic).

- Possible to subclass SLList to build a Set, but conceptually weird, e.g. get(i) doesn’t make sense, because sets are not ordered.

The Object Class

As it happens, every type in Java is a descendant of the Object class.

- VengefulSLList extends SLList.

- SLList extends Object (implicitly).

Documentation for Object class: Object (Java SE 9 & JDK 9)

Abstraction: As you'll learn later in this class, programs can get a tad confusing when they are really large. A way to make programs easier to handle is to use abstraction. Abstraction is hiding components of programs that people do not need to see. The user of the hidden methods should be able to use them without knowing how they work.

Previous
10. Inheritance II: Extends, Casting, Higher Order Functions

Next
10.2 Encapsulation

Last updated 5 months ago