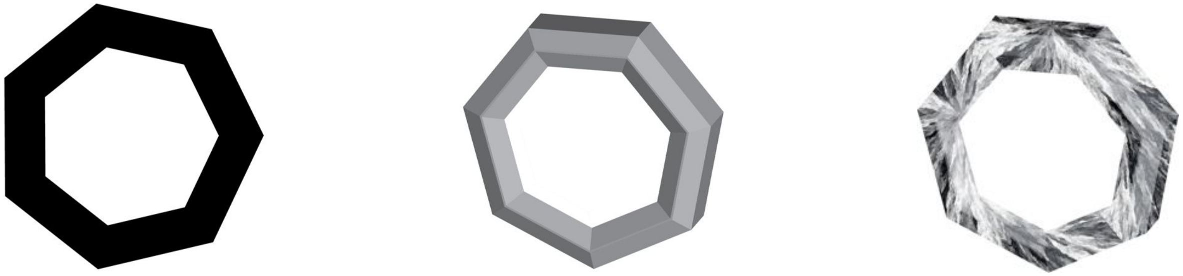


Lecture 8 (Inheritance 1)

Interface and Implementation Inheritance

CS61B, Fall 2024 @ UC Berkeley

Slides credit: Josh Hug

Note: Abridged Lecture

In FA24, we condensed the first two inheritance lectures into one.

- Because in FA24, Midterm 1 falls on a Friday (lecture day), and we wanted to cancel that day's lecture.
- Also, because in FA24, we needed to cover the last two inheritance lectures sooner, so you can start (and finish) Project 1B earlier (so you have more midterm studying time).

If you're a student in FA24, you don't need to know about the content we cut (it won't be tested).

If you're curious, you can check out the unabridged version from FA23:

- Inheritance 1: Slides, Video
- Inheritance 2: Slides, Video

The Desire for Generality

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- **The Desire for Generality**
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

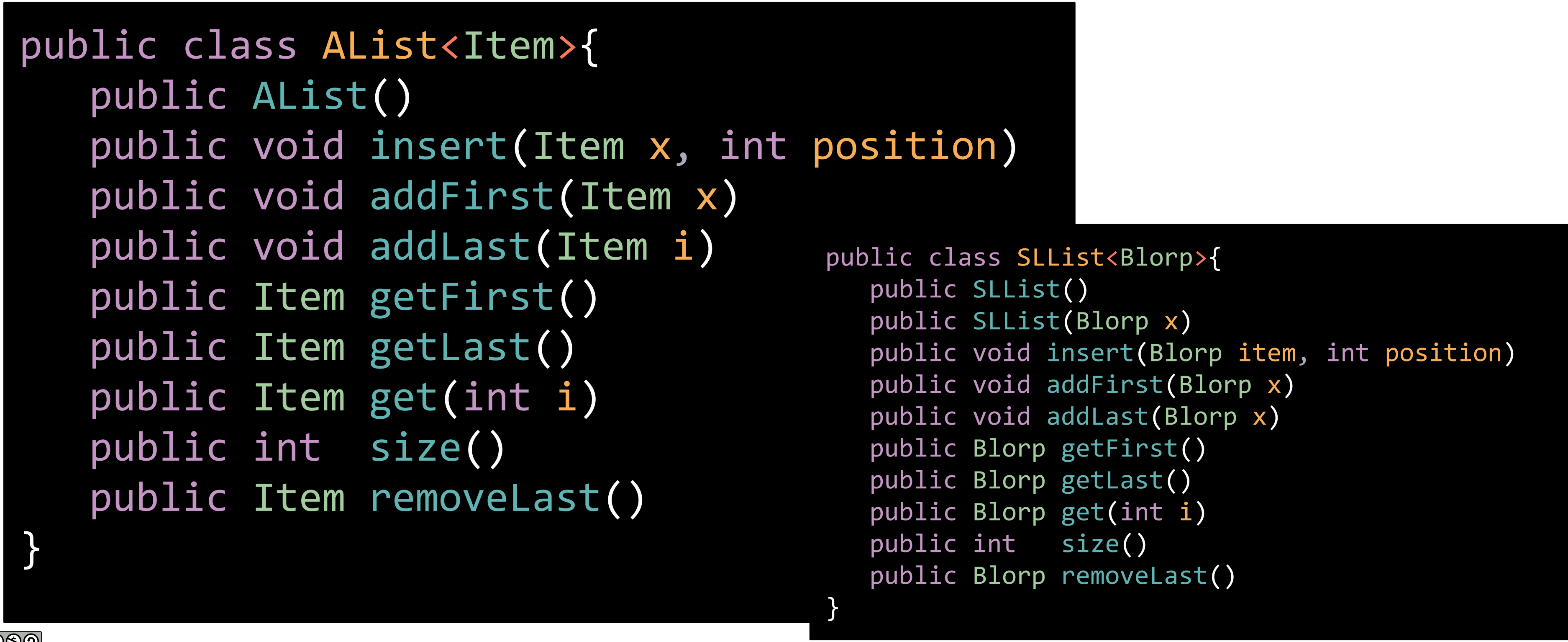
Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

Using Inheritance Safely

AList and SLList

After adding an additional “insert” method. Our AList and SLList classes from lecture have the following methods (exact same method signatures for both classes).



```
public class AList<Item>{
   public AList()
   public void insert(Item x, int position)
   public void addFirst(Item x)
   public void addLast(Item i)
   public Item getFirst()
   public Item getLast()
   public Item get(int i)
   public int  size()
   public Item removeLast()
}
```

```
public class SLList<Blorp>{
   public SLList()
   public SLList(Blorp x)
   public void insert(Blorp item, int position)
   public void addFirst(Blorp x)
   public void addLast(Blorp x)
   public Blorp getFirst()
   public Blorp getLast()
   public Blorp get(int i)
   public int   size()
   public Blorp removeLast()
}
```

Using ALists and SLLists: 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 of words:

```java
public static String longest(SLList<String> 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);
}
```

Observant viewers may note this code is very inefficient! Don't worry about it.

Demo: Using ALists and SLLists

This example usage of the `longest` method works fine.

WordUtils.java

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

public static void main(String[] args) {
   SLList<String> someList = new SLList<>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   System.out.println(longest(someList));
}
```

```
watching
```

Demo: Using ALists and SLLists

What if somebody placed their list of words in an AList instead of an SLList?

WordUtils.java

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

public static void main(String[] args) {
   AList<String> someList = new AList<>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   System.out.println(longest(someList));
}
```

AList instead of SLList.

Demo: Using ALists and SLLists

What if somebody placed their list of words in an AList instead of an SLList?

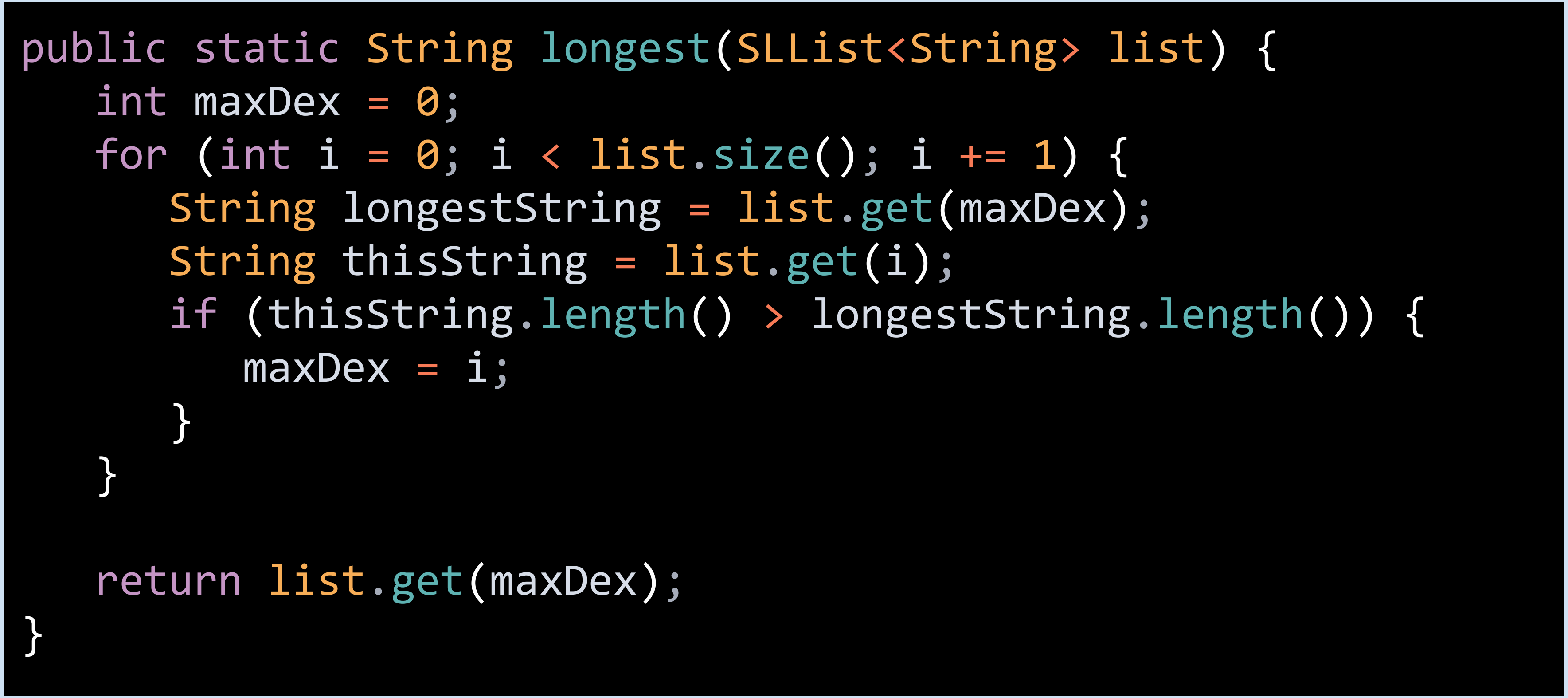
WordUtils.java

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

public static void main(String[] args) {
   AList<String> someList = new AList<>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   System.out.println(longest(someList));
}
```

Compiler error:
SLList cannot be
applied to AList.

Using ALists and SLLists: WordUtils.java

If we want longest to be able to handle ALists, what changes do we need to make?



```java
public static String longest(SLList<String> 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);
}
```

Using ALists and SLLists: WordUtils.java

If we want longest to be able to handle ALists, what changes do we need to make?

```java
public static String longest(AList<String> 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);
}
```

Method Overloading in Java

Java allows multiple methods with same name, but different parameters.

- This is called method **overloading**.

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

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

Possible solution: Copy-paste the same method body into two methods with different signatures.

The Downsides

While overloading works, it is a bad idea in the case of `longest`. Why?

- Code is virtually identical. Aesthetically gross.
- Won’t work for future lists. If we create a QList class, have to make a third method.
- Harder to **maintain**.
    - Example: Suppose you find a bug in one of the methods. You fix it in the SLList version, and forget to do it in the AList version.

Hypernyms and Hyponyms

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- **Hypernyms and Hyponyms**
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

Using Inheritance Safely

Hypernyms

In natural languages (English, Spanish, Chinese, Tagalog, etc.), we have a concept known as a “hypernym” to deal with this problem.

- Dog is a “hypernym” of poodle, malamute, yorkie, etc.

Washing your poodle:
1. Brush your poodle before a bath. ...
2. Use lukewarm water. ...
3. Talk to your poodle in a calm voice.
...
4. Use poodle shampoo. ...
5. Rinse well. ...
6. Air-dry. ...
7. Reward your poodle.

Washing your malamute:
1. Brush your malamute before a bath. ...
2. Use lukewarm water. ...
3. Talk to your malamute in a calm voice.
...
4. Use malamute shampoo. ...
5. Rinse well. ...
6. Air-dry. ...
7. Reward your malamute.

Hypernyms

In natural languages (English, Spanish, Chinese, Tagalog, etc.), we have a concept known as a "hypernym" to deal with this problem.

- Dog is a "hypernym" of poodle, malamute, yorkie, etc.

Washing your poodle:
1. Brush your poodle b
2. Use lukewarm wate
3. Talk to your poodle i
...
4. Use poodle shampo
5. Rinse well. ...
6. Air-dry. ...
7. Reward your poodle

Washing your **dog**:
1. Brush your **dog** before a bath. ...
2. Use lukewarm water. ...
3. Talk to your **dog** in a calm voice. ...
4. Use dog shampoo. ...
5. Rinse well. ...
6. Air-dry. ...
7. Reward your **dog**.

mute:
mute before a bath. ...
ater. ...
amute in a calm voice.
hampoo. ...
lamute.

Hypernym and Hyponym

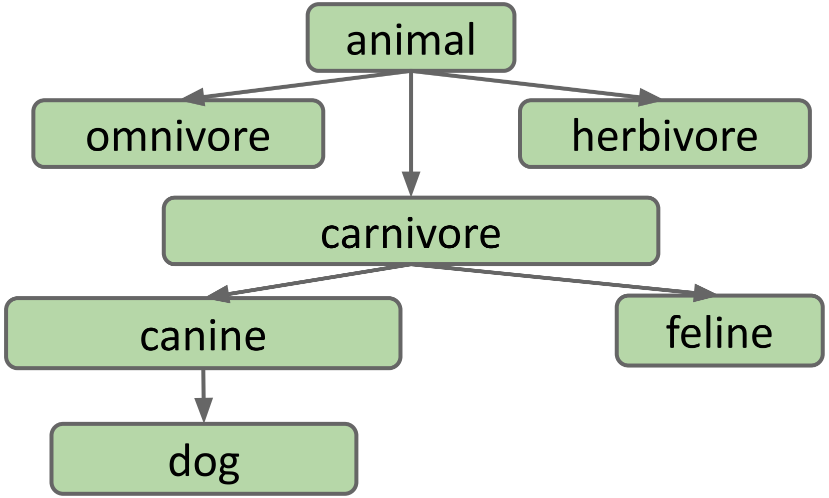
We use the word hyponym for the opposite type of relationship.

- “dog”: Hypernym of “poodle”, “malamute”, “dachshund”, etc.
- “poodle”: Hyponym of “dog”

Hypernyms and hyponyms comprise a hierarchy.

- A dog “is-a” canine.
- A canine “is-a” carnivore.
- A carnivore “is-an” animal.

(for fun: see the WordNet project)



Interface and Implements Keywords

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- **Interface and Implements Keywords**
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

Using Inheritance Safely

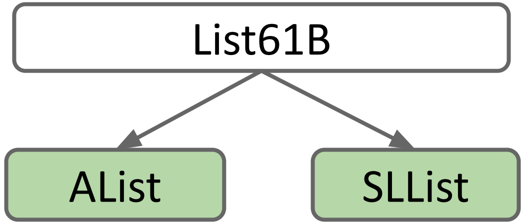
Simple Hyponymic Relationships in Java

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:

- Step 1: Define a reference type for our hypernym (List61B.java).
- Step 2: Specify that SLLists and ALists are hyponyms of that type.



Step 1: Defining a List61B

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.

Step 1: Defining a List61B

We'll use the new keyword **interface** instead of **class** to define a List61B.

- Idea: Interface is a specification of **<u>what</u>** a List is able to do, **<u>not how</u>** to do it.

List61B.java

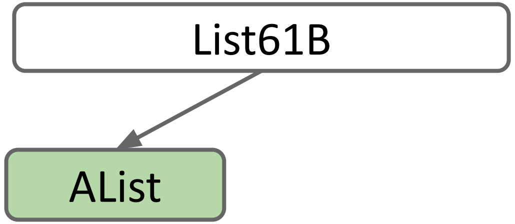
```java
public interface List61B<Item> {
   public void insert(Item x, int position);
   public void addFirst(Item x);
   public void addLast(Item y);
   public Item getFirst();
   public Item getLast();
   public Item removeLast();
   public Item get(int i);
   public int  size();
}
```

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 void addLast(Item x) {
      ...
```

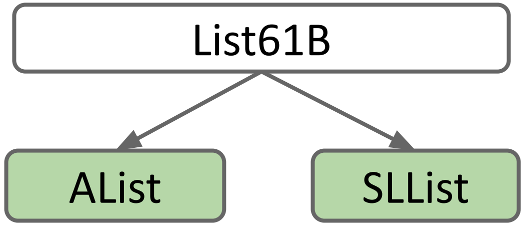


Step 2: Implementing the List61B Interface

We’ll now:

- Use the new **implements** keyword to tell the Java compiler that SLList and AList are hyponyms of List61B.

```
public class SLList<Blorp> implements List61B<Blorp>{
   ...
   public void addLast(Blorp x) {
      ...
```



Adjusting WordUtils.java

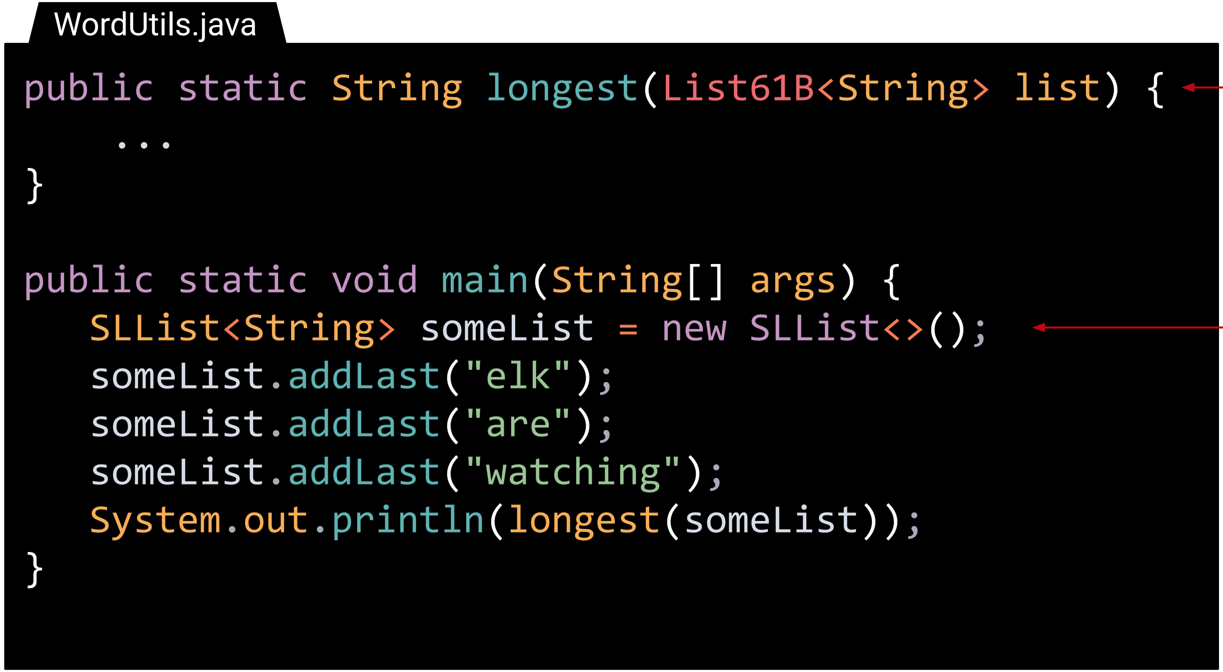
We can now adjust our longest method to work on either kind of list:

```
public static String longest(List61B<String> 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);
}
```

```
AList<String> a = new AList<>();
a.addLast("egg");
a.addLast("boyz");
longest(a);
```

Demo: Interface and Implements Keywords

Our `longest` method now takes in a List61B (not a SLList or AList).

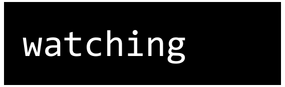


WordUtils.java

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

public static void main(String[] args) {
   SLList<String> someList = new SLList<>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   System.out.println(longest(someList));
}
```

You can pass in any object that implements List61B…

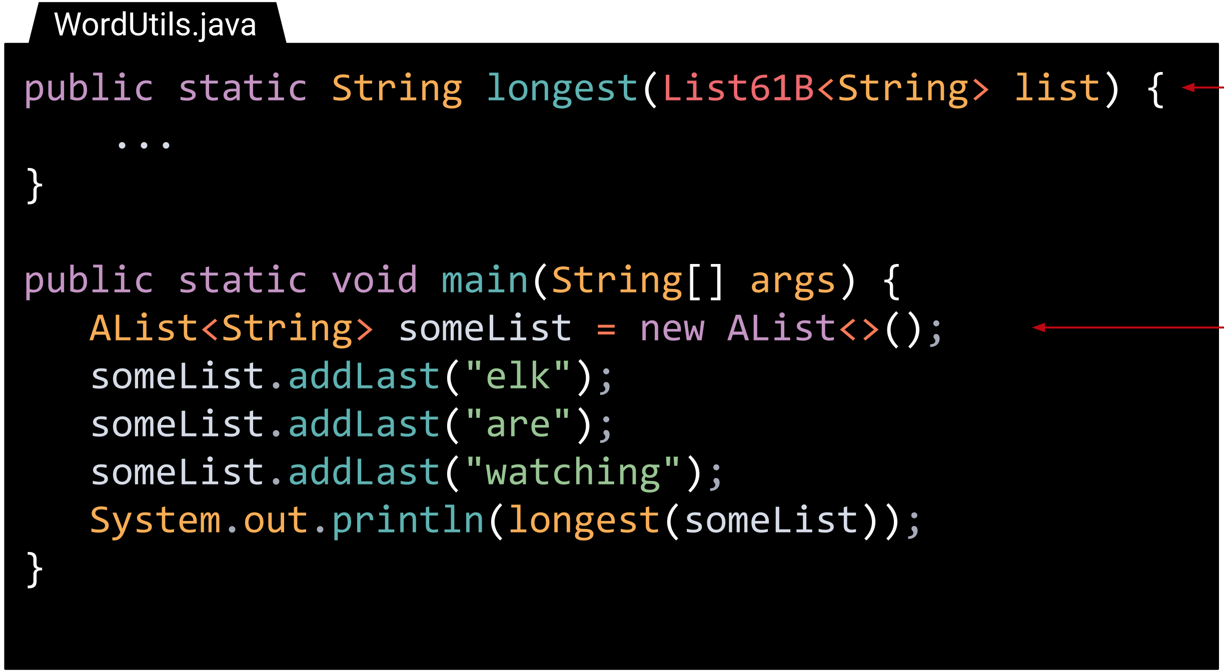
…including SLList.



```
watching
```

Demo: Interface and Implements Keywords

Our `longest` method now takes in a List61B (not a SLList or AList).



WordUtils.java

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

public static void main(String[] args) {
   AList<String> someList = new AList<>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   System.out.println(longest(someList));
}
```

You can pass in any object that implements List61B…

…including AList.

```
watching
```

Overriding vs. Overloading

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- **Overriding vs. Overloading**
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

Using Inheritance Safely

Method Overriding

If a “subclass” has a method with the exact same signature as in the “superclass”, we say the subclass **overrides** the method.

```
public interface List61B<Item> {
   public void addLast(Item y);
   ...
```

```
public class AList<Item> implements List61B<Item>{
   ...
   public void addLast(Item x) {
      ...
```

AList overrides addLast(Item)

Method Overriding vs. Overloading

If a “subclass” has a method with the exact same signature as in the “superclass”, we say the subclass **overrides** the method.

- `Animal`’s subclass `Pig` overrides the `makeNoise()` method.

- Methods with the same name but different signatures are **overloaded**.

```
public interface Animal {
   public void makeNoise();
}
```

```
public class Pig implements Animal {
   public void makeNoise() {
      System.out.print("oink");
   }
}
```

`Pig` overrides `makeNoise()`

```
public class Dog implements Animal {
   public void makeNoise(Dog x)
   public void makeNoise()
```

`makeNoise` is overloaded

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

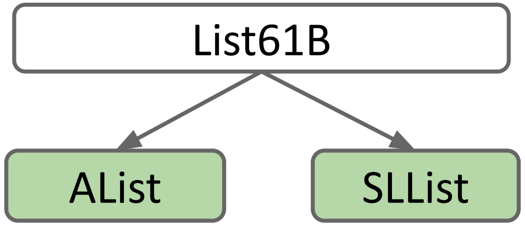
`abs` is overloaded

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.

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

   @Override
   public void addLast(Item x) {
      ...
```



Method Overriding

If a subclass has a method with the exact same signature as in the superclass, we say the subclass **overrides** the method.

- Even if you don't write @Override, subclass still overrides the method.
- @Override is just an optional reminder that you're overriding.

Why use @Override?

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

Interface Inheritance

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- **Interface Inheritance**

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

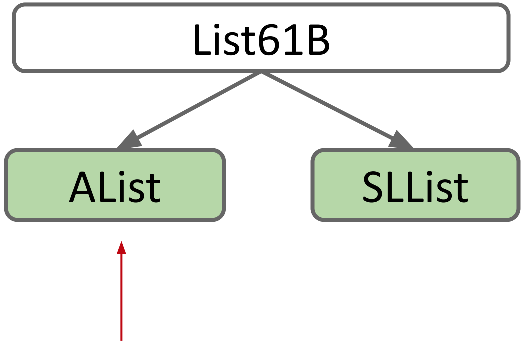
Using Inheritance Safely

Interface Inheritance

Specifying the capabilities of a subclass using the **implements** keyword is known as **<u>interface inheritance</u>**.

- Interface: The list of all method signatures.
- Inheritance: The subclass “inherits” the interface.
- Specifies what the subclass can do, but not how.
- Subclasses <u>must</u> override all of these methods!
    - Will fail to compile otherwise.

```
public interface List61B<Item> {
   public void addFirst(Item x);
   ...
   public void proo();
}
```

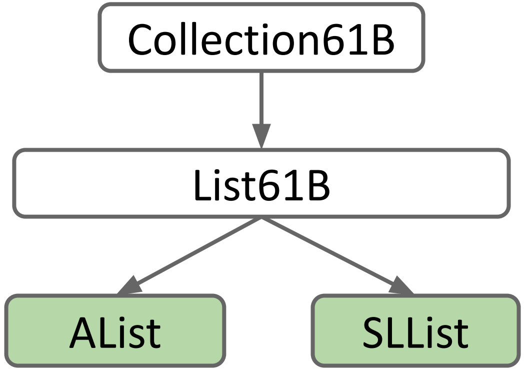


If `AList` doesn’t have a `proo()` method, `AList` will not compile!

Interface Inheritance

Specifying the capabilities of a subclass using the **implements** keyword is known as **<u>interface inheritance</u>**.

- Interface: The list of all method signatures.
- Inheritance: The subclass “inherits” the interface.
- Specifies what the subclass can do, but not how.
- Subclasses <u>must</u> override all of these methods!
- Such relationships can be multi-generational.
  - Figure: Interfaces in white, classes in green.
  - We’ll talk about this in a later lecture.



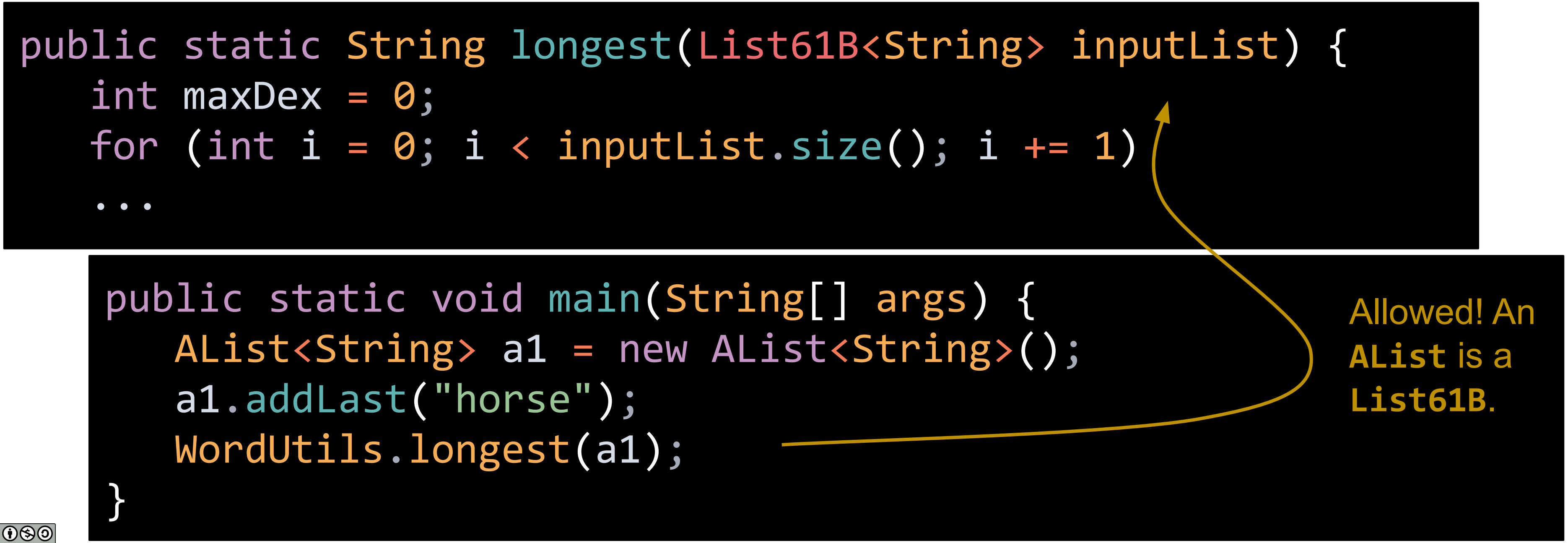
Interface inheritance is a powerful tool for generalizing code.

- `WordUtils.longest` works on SLLists, ALists, and even lists that have not yet been invented!

Is-a-relationships

Recall: A memory box can only hold 64 bit addresses for the appropriate type.

- Example: **`inputList`** can only hold a **`List61B<String>`**.
- An **`AList`** is-a **`List61B`**, so **`inputList`** can hold a reference to the **`AList`**.



```java
public static String longest(List61B<String> inputList) {
   int maxDex = 0;
   for (int i = 0; i < inputList.size(); i += 1)
   ...
```

```java
public static void main(String[] args) {
   AList<String> a1 = new AList<String>();
   a1.addLast("horse");
   WordUtils.longest(a1);
}
```

Allowed! An **`AList`** is a **`List61B`**.

Question

Will the code below compile? If so, what happens when it runs?

a. Will not compile.

b. Will compile, but will cause an error at runtime on the **new** line.

c. When it runs, an **`SLList`** is created and its address is stored in the **`someList`** variable, but it crashes on **`someList.addFirst()`** since the **`List`** interface doesn't implement **`addFirst`**.

d. When it runs, an **`SLList`** is created and its address is stored in the **`someList`** variable. Then the string "elk" is inserted into the **`SLList`** referred to by **`addFirst`**.

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

Question

Will the code below compile? If so, what happens when it runs?

a. Will not compile.

b. Will compile, but will cause an error at runtime on the **new** line.

c. When it runs, an **`SLList`** is created and its address is stored in the **`someList`** variable, but it crashes on **`someList.addFirst()`** since the **`List`** interface doesn't implement **`addFirst`**.

d. **When it runs, an `SLList` is created and its address is stored in the `someList` variable. Then the string "elk" is inserted into the `SLList` referred to by `addFirst`.**

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

Extends Keyword: Rotating SLList

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- **Extends Keyword: Rotating SLList**
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

Using Inheritance Safely

Implementation Inheritance

Interface inheritance:

- Subclass inherits signatures, but NOT implementation.

For better or worse, Java also allows **implementation inheritance**.

- Subclasses can inherit signatures AND implementation.

Use the **extends** keyword to inherit methods from a **class.**

- Unlike the **implements** keyword, which inherits signatures from an **interface**.

The Extends Keyword

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

- Example: `SLList<Blorp>` **`implements`** `List61B<Blorp>`

instead of an interface

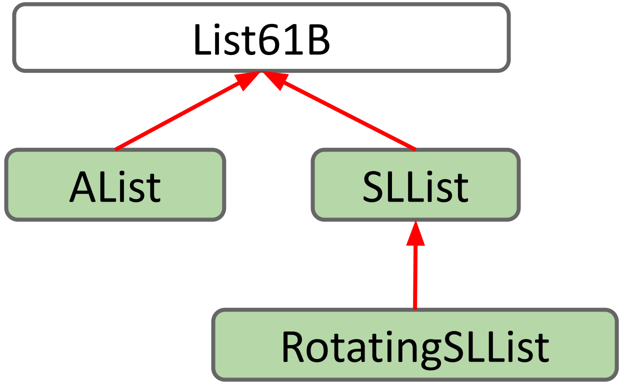
If you want one class to be a hyponym of another *class*, you use **extends.**

We'd like to build RotatingSLList that can perform any SLList operation as well as:

- rotateRight(): Moves back item the front.

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

- After rotateRight: [22, 5, 9, 15]



Demo: Rotating SLList

RotatingSLList.java

```java
public class RotatingSLList<Item> {

   public static void main(String[] args) {
      RotatingSLList<Integer> rsl = new RotatingSLList<>();
      /* Creates SList: [10, 11, 12, 13] */
      rsl.addLast(10);
      rsl.addLast(11);
      rsl.addLast(12);
      rsl.addLast(13);

      /* Should be: [13, 10, 11, 12] */
      rsl.rotateRight();
      rsl.print();
   }
}
```

This does not compile. The RotatingSLList is missing the addLast, rotateRight, and print methods.

Demo: Rotating SLList

RotatingSLList.java

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

   public static void main(String[] args) {
      RotatingSLList<Integer> rsl = new RotatingSLList<>();
      /* Creates SList: [10, 11, 12, 13] */
      rsl.addLast(10);
      rsl.addLast(11);
      rsl.addLast(12);
      rsl.addLast(13);

      /* Should be: [13, 10, 11, 12] */
      rsl.rotateRight();
      rsl.print();
   }
}
```

Now the compiler knows that a RotatingSLList is a SLList, so RotatingSLList can inherit the addLast and print methods from the SLList class.

The rotateRight method is still missing.

Demo: Rotating SLList

RotatingSLList.java

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

   /** Rotates list to the right. */
   public void rotateRight() {


   }

}
```

Demo: Rotating SLList

RotatingSLList.java

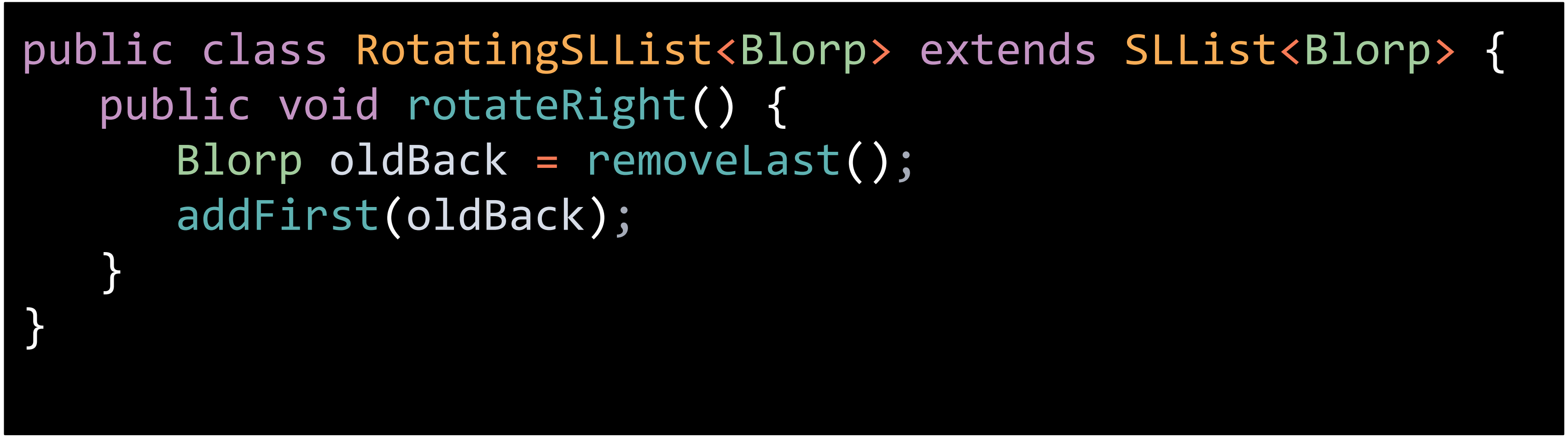
```java
public class RotatingSLList<Item> extends SLList<Item> {

   /** Rotates list to the right. */
   public void rotateRight() {
      Item x = removeLast();

   }

}
```

Demo: Rotating SLList

RotatingSLList.java

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

   /** Rotates list to the right. */
   public void rotateRight() {
      Item x = removeLast();
      addFirst(x);
   }

}
```

RotatingSLList



```
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.

… but members may be private and thus inaccessible! More later.

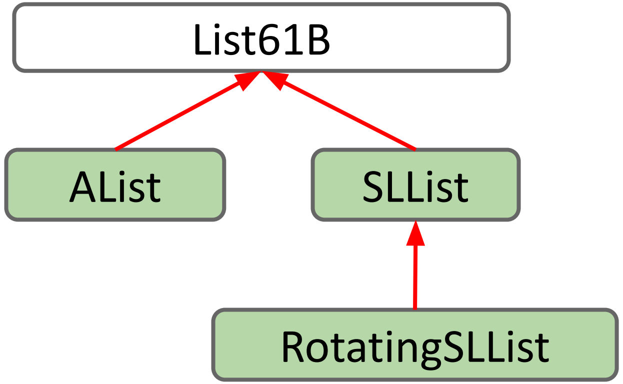
Constructors are not inherited.

Clarification: Implements vs. Extends

How do you know which to pick between “implements” and “extends”?

- You must use “implements” if the hypernym is an interface and the hyponym is a class (e.g. hypernym List, hyponym AList).
- You must use “extends” in all other cases.

There’s no choice that you have to make, the Java designers just picked a different keyword for the two cases.



Super Keyword: Vengeful SLList

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- **Super Keyword: Vengeful SLList**
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

Using Inheritance Safely

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 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. */
}
```

Coding Demo: Vengeful SLList

VengefulSLList.java

```java
public class VengefulSLList<Item> extends SLList<Item> {









    public void printLostItems() {

    }
}
```

Coding Demo: Vengeful SLList

VengefulSLList.java

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









    public void printLostItems() {

    }
}
```

Coding Demo: Vengeful SLList

VengefulSLList.java

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









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

Coding Demo: Vengeful SLList

VengefulSLList.java

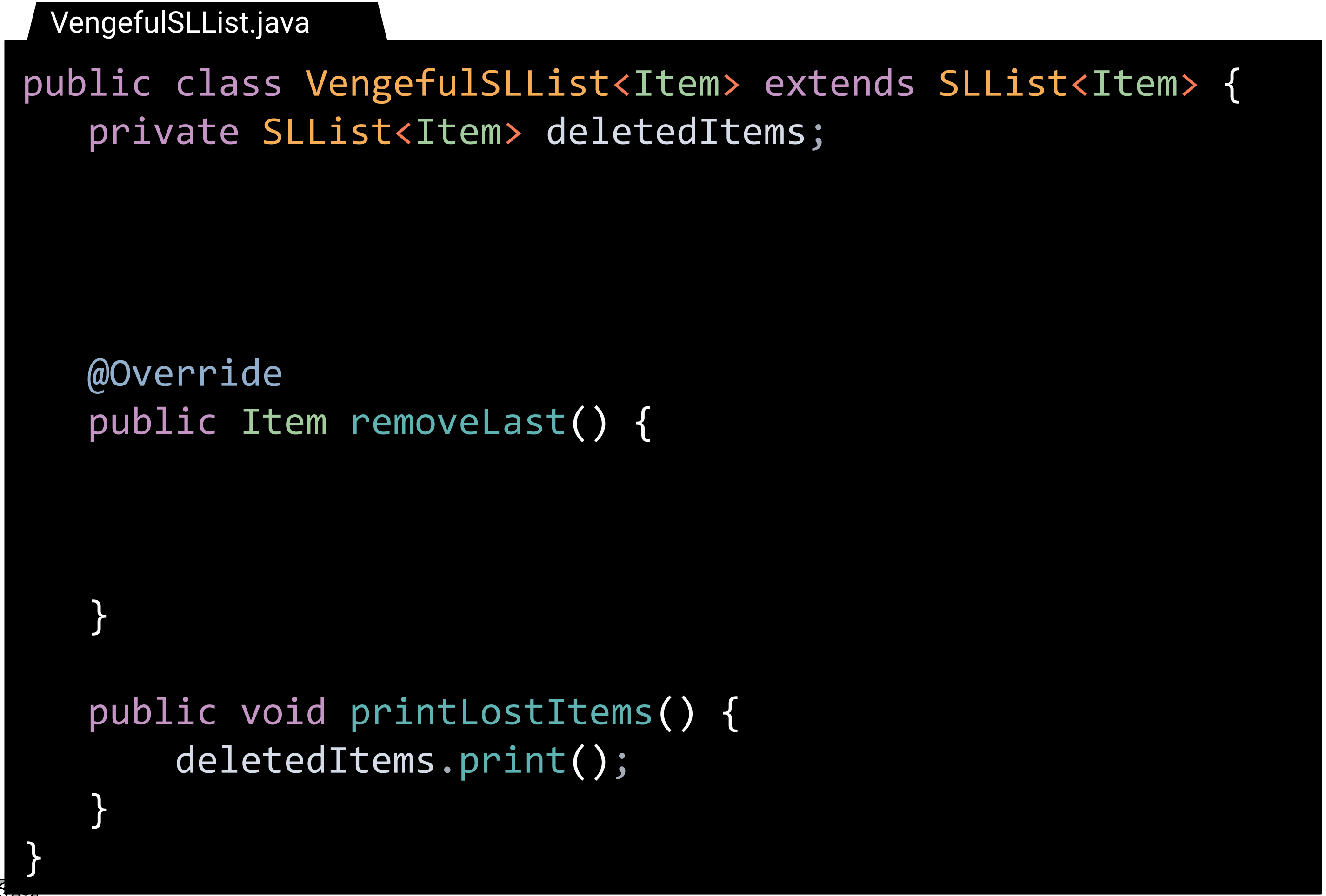
```java
public class VengefulSLList<Item> extends SLList<Item> {
    private SLList<Item> deletedItems;





    public Item removeLast() {



    }

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

Coding Demo: Vengeful SLList

VengefulSLList.java

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





   @Override
   public Item removeLast() {




   }

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

Coding Demo: Vengeful SLList



VengefulSLList.java

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




   @Override
   public Item removeLast() {



   }

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

We could try to copy-paste the removeLast method from SLList.

Problem: SLList's removeLast method uses private variables like sentinel and size. VengefulSLList cannot access these variables.

Coding Demo: Vengeful SLList

VengefulSLList.java

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




    @Override
    public Item removeLast() {
        Item x = super.removeLast();


    }

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

Solution: Use the super keyword to call SLList's removeLast method.

Coding Demo: Vengeful SLList

VengefulSLList.java

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




   @Override
   public Item removeLast() {
       Item x = super.removeLast();
       deletedItems.addLast(x);

   }

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

Coding Demo: Vengeful SLList

VengefulSLList.java

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





   @Override
   public Item removeLast() {
       Item x = super.removeLast();
       deletedItems.addLast(x);
       return x;
   }

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

Coding Demo: Vengeful SLList

VengefulSLList.java

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





   @Override
   public Item removeLast() {
       Item x = super.removeLast();
       deletedItems.addLast(x);
       return x;
   }

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

If we run this, we get an exception.

deletedItems is null. It was never initialized (we never created an actual list), so we can't add to deletedItems.

Coding Demo: Vengeful SLList

VengefulSLList.java

```java
public class VengefulSLList<Item> extends SLList<Item> {
   private SLList<Item> deletedItems;
   public VengefulSLList() {

   }

   @Override
   public Item removeLast() {
       Item x = super.removeLast();
       deletedItems.addLast(x);
       return x;
   }

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

Solution: Add a constructor that initializes the deletedItems list.

Note: You could also initialize the list on the same line you declared the deletedItems variable.

Coding Demo: Vengeful SLList

VengefulSLList.java

```java
public class VengefulSLList<Item> extends SLList<Item> {
   private SLList<Item> deletedItems;
   public VengefulSLList() {
       deletedItems = new SLList<Item>();
   }

   @Override
   public Item removeLast() {
       Item x = super.removeLast();
       deletedItems.addLast(x);
       return x;
   }

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

Another Example: VengefulSLList

```
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();
       deletedItems.addLast(oldBack);
       return oldBack;
   }

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

calls Superclass's version of removeLast()

Note: Java syntax disallows super.super. For a nice description of why, see this link.

A Boring Constructor Gotcha

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

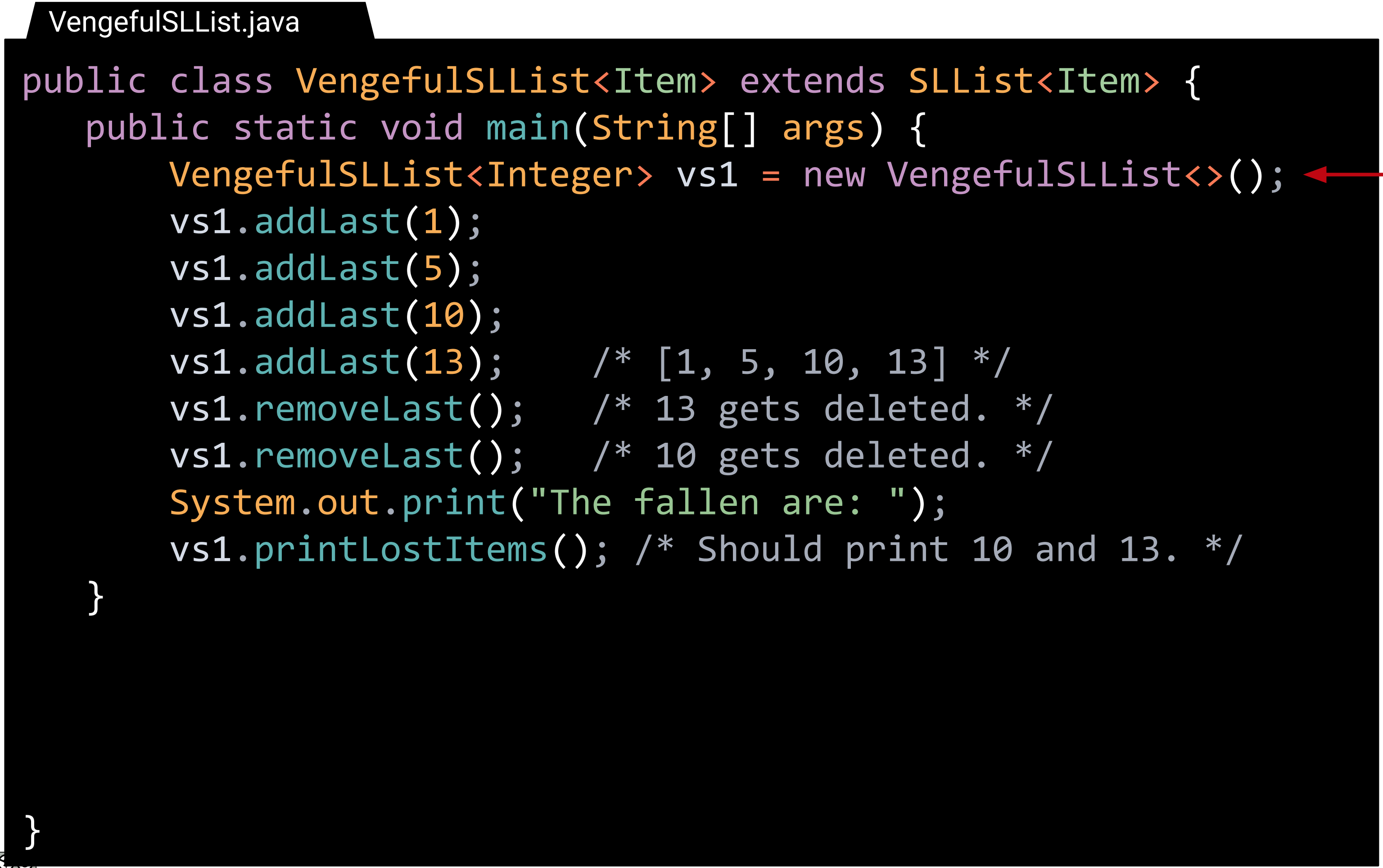
Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- **A Boring Constructor Gotcha**
- The Object Class

Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

Coding Demo: Vengeful SLList



VengefulSLList.java

```java
public class VengefulSLList<Item> extends SLList<Item> {
   public static void main(String[] args) {
       VengefulSLList<Integer> vs1 = new VengefulSLList<>();
       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. */
   }



}
```

Set a breakpoint here.

Then step *in* (not *over*).

Coding Demo: Vengeful SLList

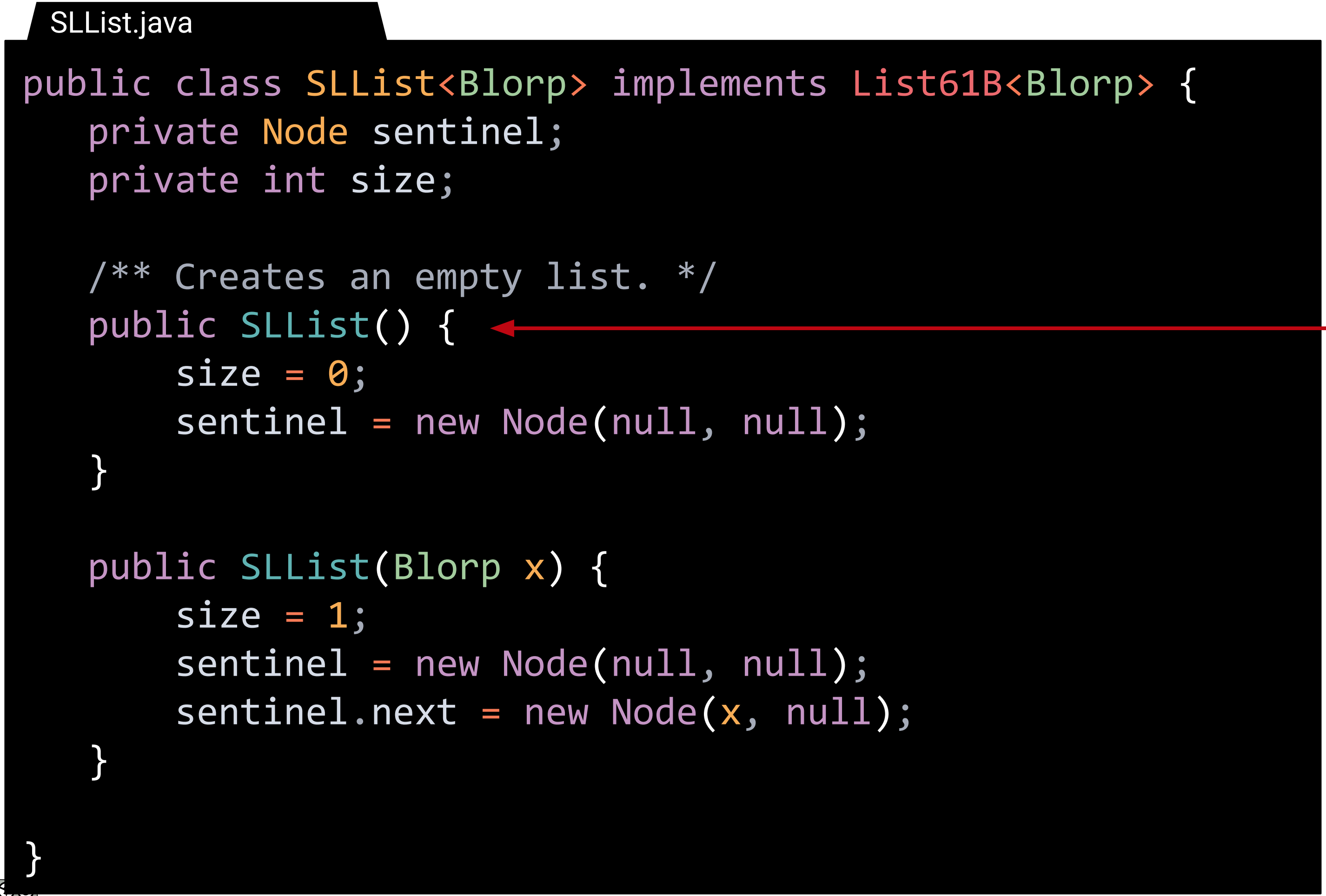
VengefulSLList.java

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

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










}
```

We step into the VengefulSLList constructor.

Then step *in* again (not *over*).

Coding Demo: Vengeful SLList



SLList.java

```java
public class SLList<Blorp> implements List61B<Blorp> {
    private Node sentinel;
    private int size;

    /** Creates an empty list. */
    public SLList() {
        size = 0;
        sentinel = new Node(null, null);
    }

    public SLList(Blorp x) {
        size = 1;
        sentinel = new Node(null, null);
        sentinel.next = new Node(x, null);
    }

}
```

We step into the constructor of SLList (the super class).

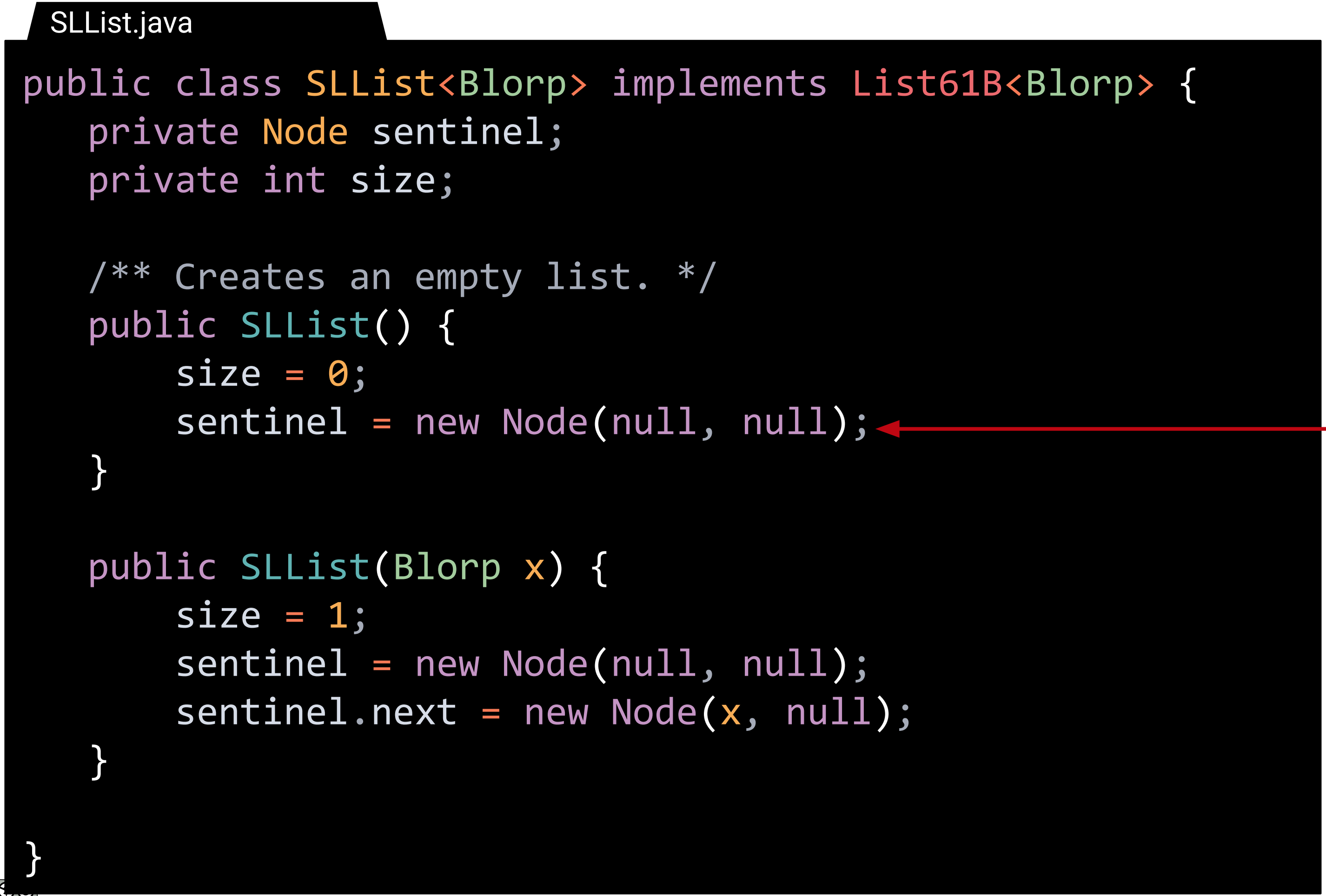
Coding Demo: Vengeful SLList

SLList.java

```java
public class SLList<Blorp> implements List61B<Blorp> {
   private Node sentinel;
   private int size;

   /** Creates an empty list. */
   public SLList() {
       size = 0;
       sentinel = new Node(null, null);
   }

   public SLList(Blorp x) {
       size = 1;
       sentinel = new Node(null, null);
       sentinel.next = new Node(x, null);
   }

}
```

This helps us correctly set up size…

Coding Demo: Vengeful SLList

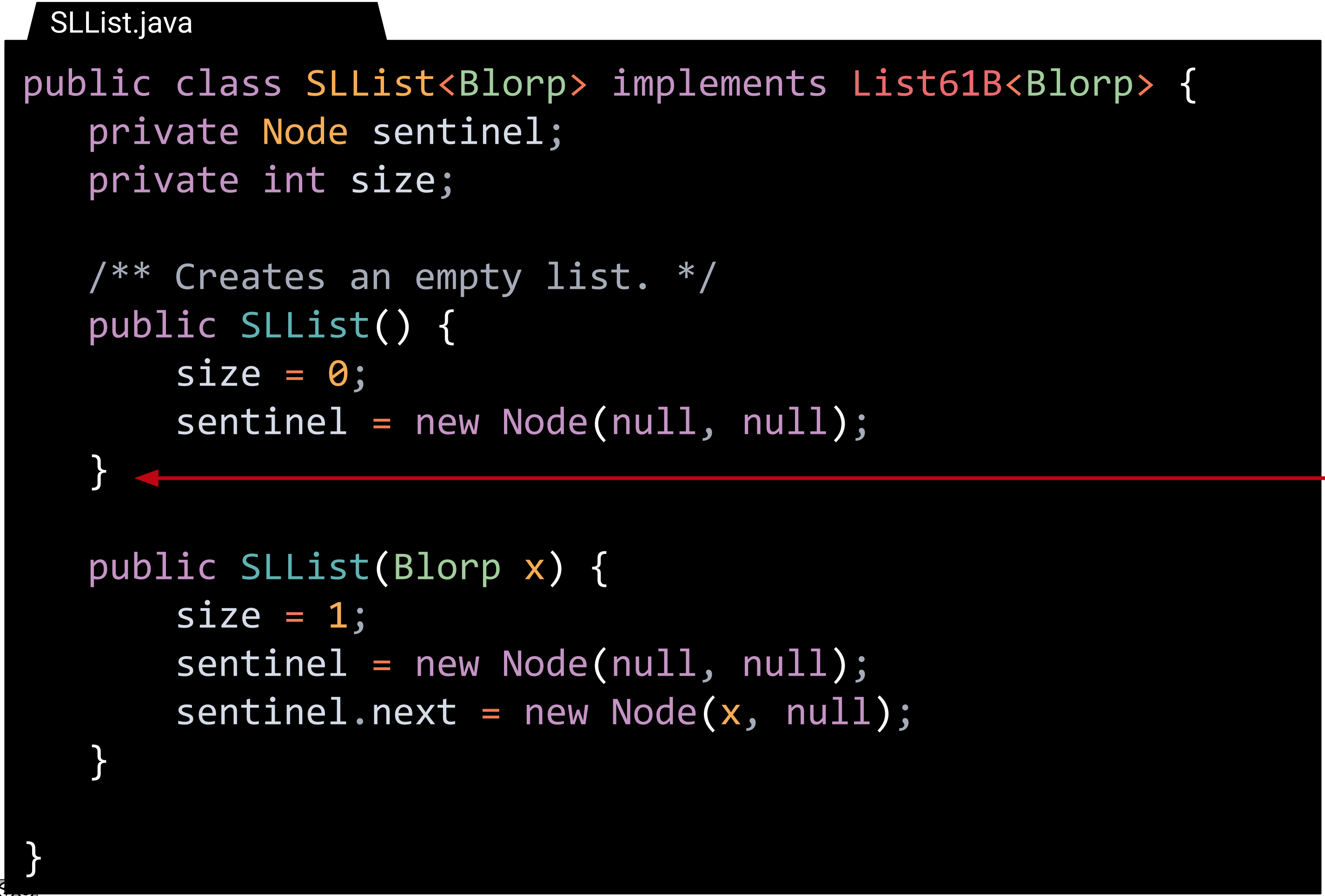


SLList.java

```java
public class SLList<Blorp> implements List61B<Blorp> {
   private Node sentinel;
   private int size;

   /** Creates an empty list. */
   public SLList() {
       size = 0;
       sentinel = new Node(null, null);
   }

   public SLList(Blorp x) {
       size = 1;
       sentinel = new Node(null, null);
       sentinel.next = new Node(x, null);
   }

}
```

…and correctly set up sentinel.

Coding Demo: Vengeful SLList

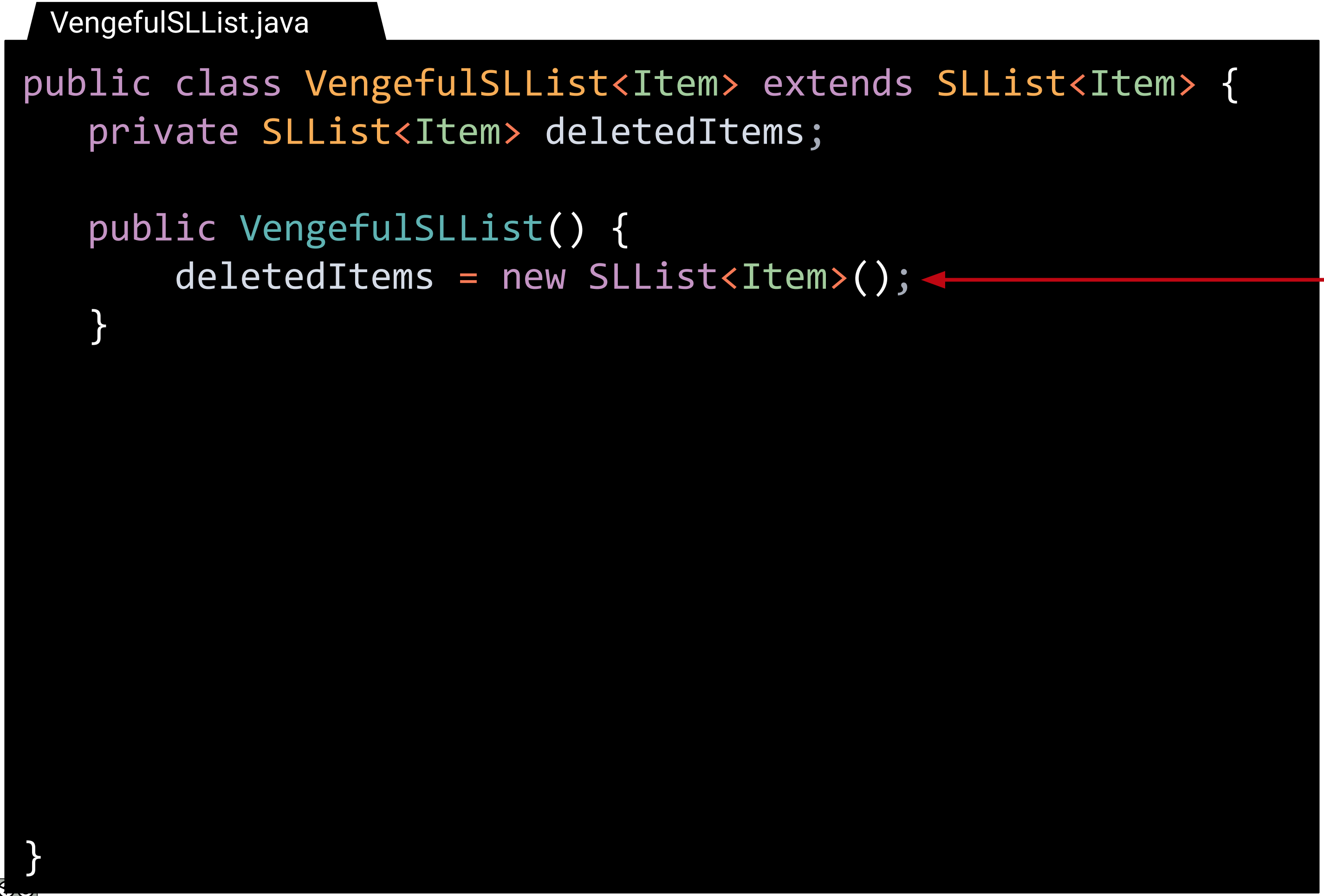


SLList.java

```
public class SLList<Blorp> implements List61B<Blorp> {
   private Node sentinel;
   private int size;

   /** Creates an empty list. */
   public SLList() {
       size = 0;
       sentinel = new Node(null, null);
   }

   public SLList(Blorp x) {
       size = 1;
       sentinel = new Node(null, null);
       sentinel.next = new Node(x, null);
   }

}
```

Then we'll return back to the VengefulSLList constructor we came from.

Coding Demo: Vengeful SLList



VengefulSLList.java

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

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









}
```

Back out to the VengefulSLList constructor.

Here, we'll finish setting up the deletedItems list, which is specific to the child class.

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 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 <u>automatically</u> do it for you.

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

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

must come first!

These constructors are exactly equivalent.

Calling Other Constructors

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

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

calls SLList(Item x)

Not equivalent! Code to the right makes implicit call to super(), not super(x).

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

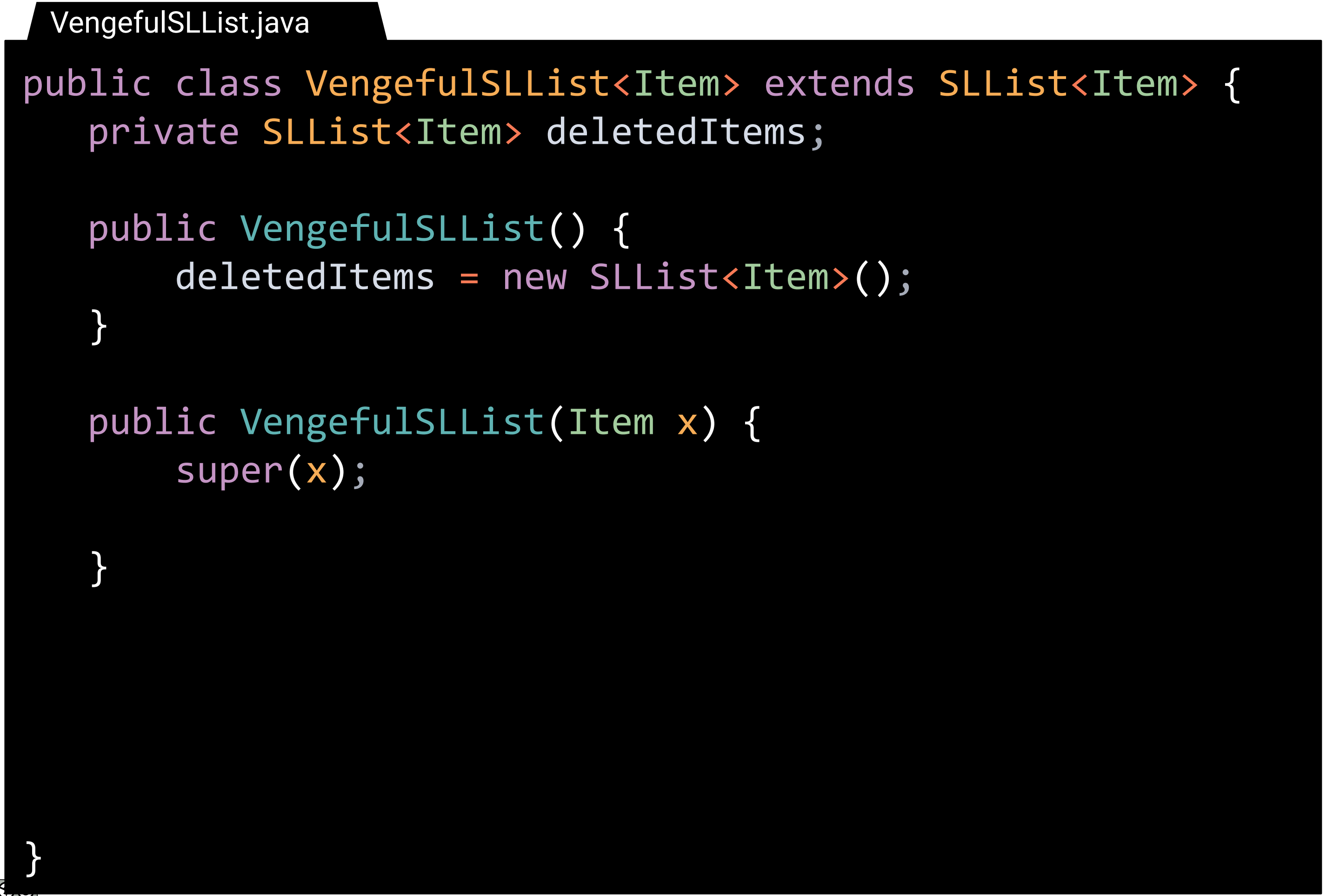
Coding Demo: Vengeful SLList

VengefulSLList.java

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

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

   public VengefulSLList(Item x) {


   }






}
```

Let's write a second constructor for VengefulSLList that takes in an item.

Coding Demo: Vengeful SLList

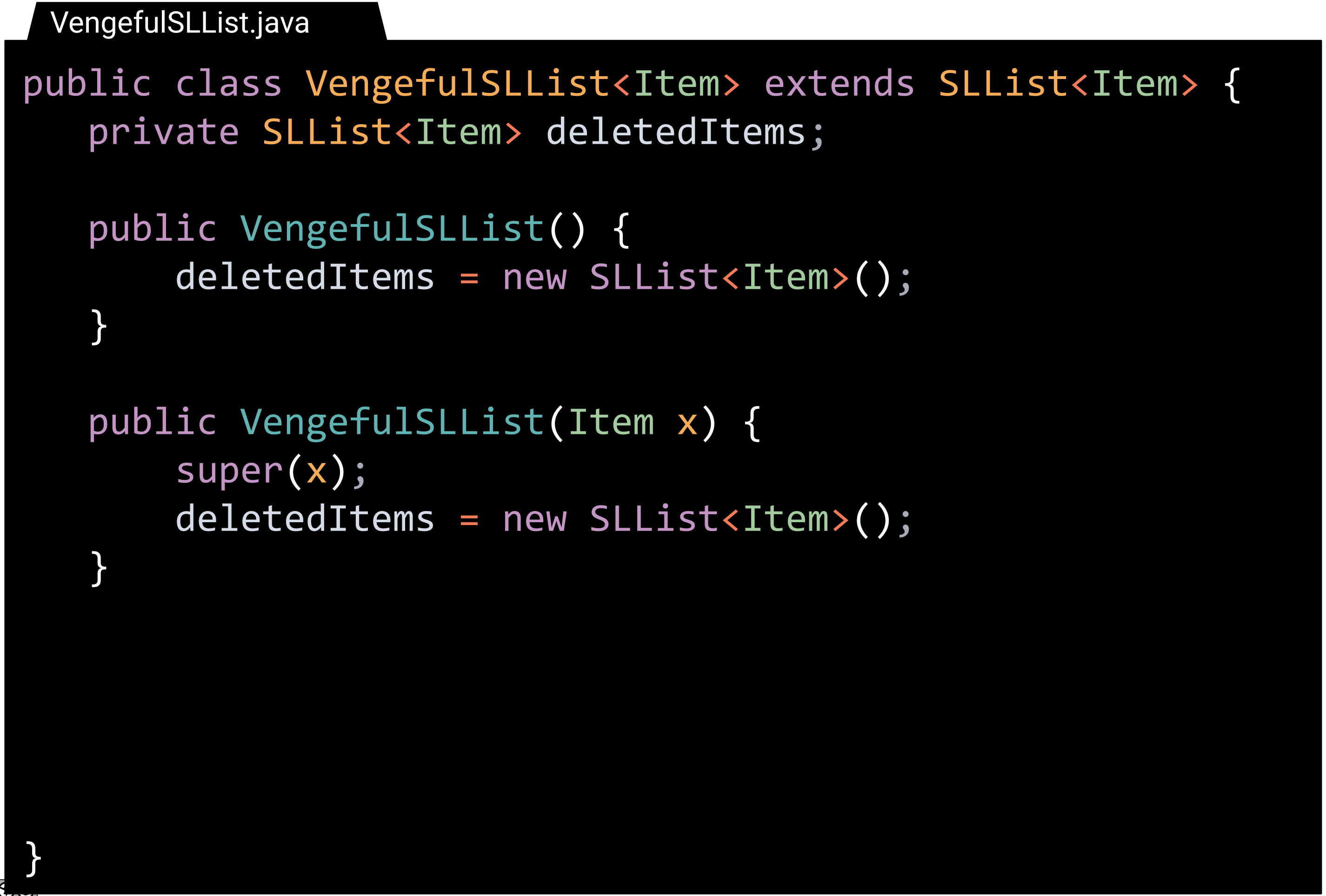


VengefulSLList.java

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

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

   public VengefulSLList(Item x) {
       super(x);

   }




}
```

Let's write a second constructor for VengefulSLList that takes in an item.

Coding Demo: Vengeful SLList



VengefulSLList.java

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

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

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




}
```

Let's write a second constructor for VengefulSLList that takes in an item.

Coding Demo: Vengeful SLList

VengefulSLList.java

```java
public class VengefulSLList<Item> extends SLList<Item> {
   public static void main(String[] args) {
       VengefulSLList<Integer> vs1 = new VengefulSLList<>(0);
       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. */
   }



}
```

Set a breakpoint here.

Then step *in* (not *over*).

Coding Demo: Vengeful SLList

VengefulSLList.java

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

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

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




}
```

We step into the VengefulSLList constructor with one argument.

Then step *in* again (not *over*).

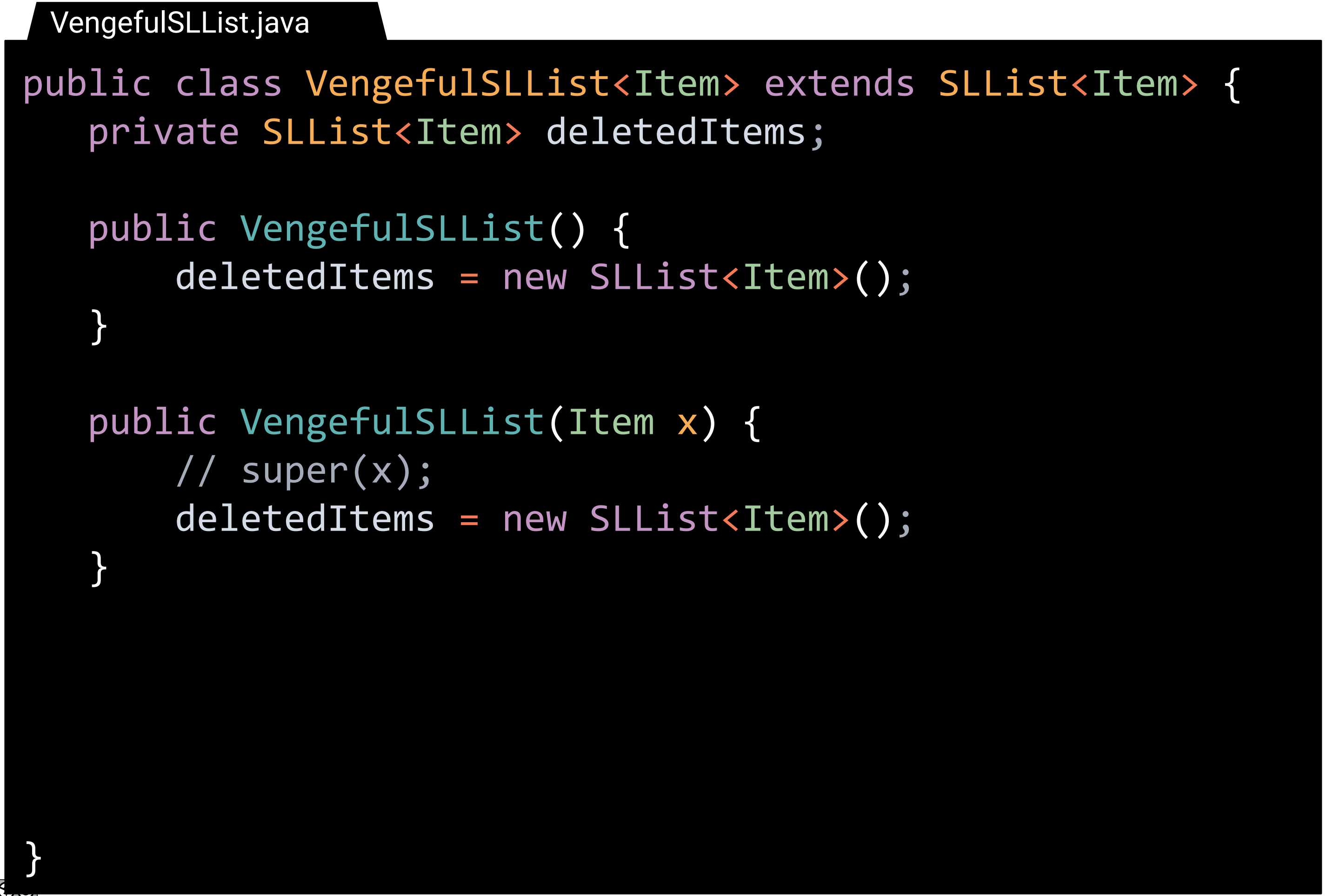
Coding Demo: Vengeful SLList

SLList.java

```java
public class SLList<Blorp> implements List61B<Blorp> {
   private Node sentinel;
   private int size;

   /** Creates an empty list. */
   public SLList() {
       size = 0;
       sentinel = new Node(null, null);
   }

   public SLList(Blorp x) {
       size = 1;
       sentinel = new Node(null, null);
       sentinel.next = new Node(x, null);
   }

}
```

We step into the SLList constructor with one argument.

Coding Demo: Vengeful SLList



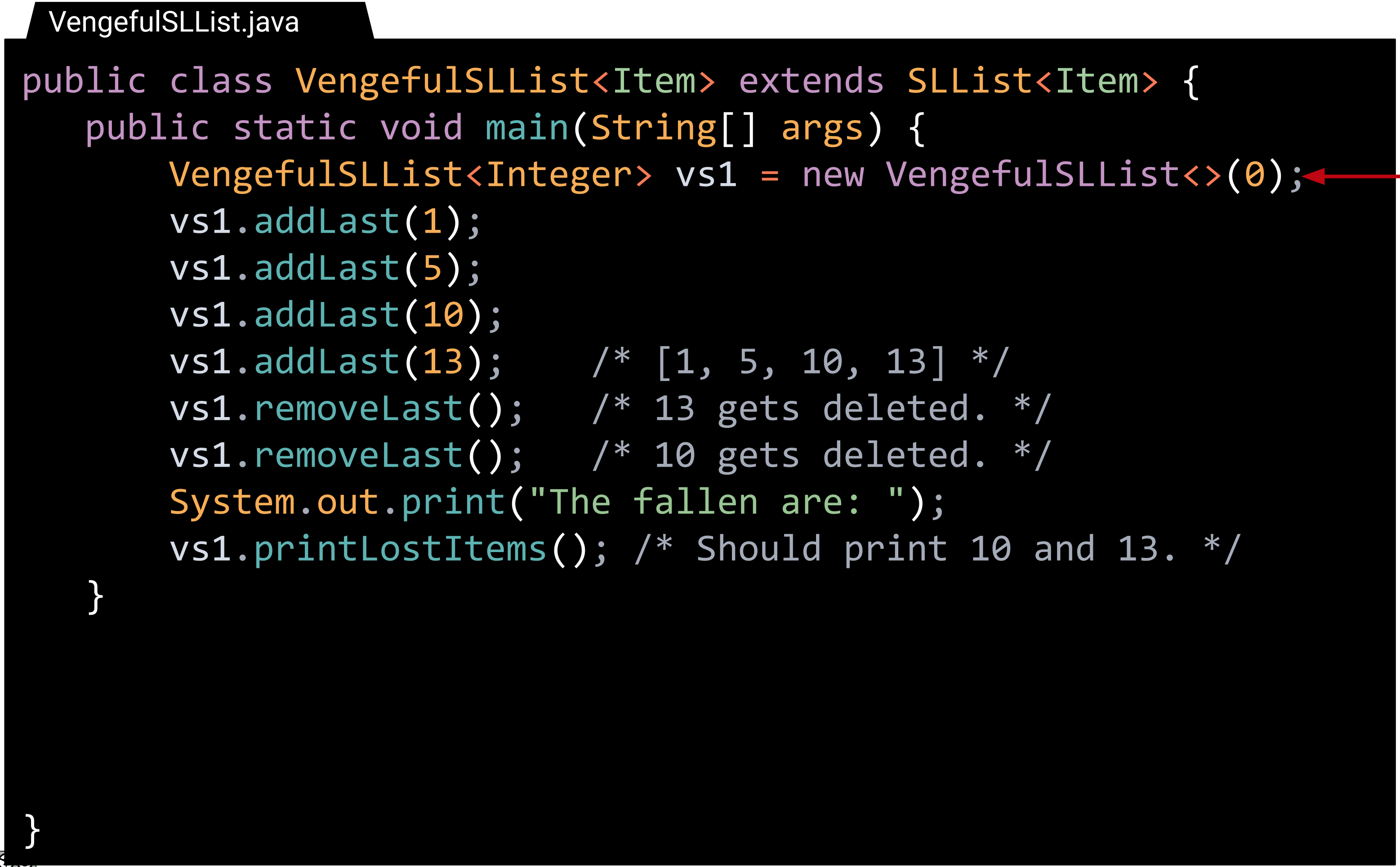
VengefulSLList.java

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

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

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




}
```

What if we didn't call the constructor?

Java still calls the no-argument constructor implicitly.

Coding Demo: Vengeful SLList



VengefulSLList.java

```java
public class VengefulSLList<Item> extends SLList<Item> {
   public static void main(String[] args) {
       VengefulSLList<Integer> vs1 = new VengefulSLList<>(0);
       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. */
   }



}
```

Set a breakpoint here.

Then step *in* (not *over*).

Coding Demo: Vengeful SLList

VengefulSLList.java

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

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

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



}
```

We step into the VengefulSLList constructor with one argument.

Then step *in* again (not *over*).

Coding Demo: Vengeful SLList

SLList.java

```java
public class SLList<Blorp> implements List61B<Blorp> {
   private Node sentinel;
   private int size;

   /** Creates an empty list. */
   public SLList() {
       size = 0;
       sentinel = new Node(null, null);
   }

   public SLList(Blorp x) {
       size = 1;
       sentinel = new Node(null, null);
       sentinel.next = new Node(x, null);
   }

}
```

Because we didn't explicitly call super, we step into the default no-argument SLList constructor.

The Object Class

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- **The Object Class**

Calling Overridden Methods

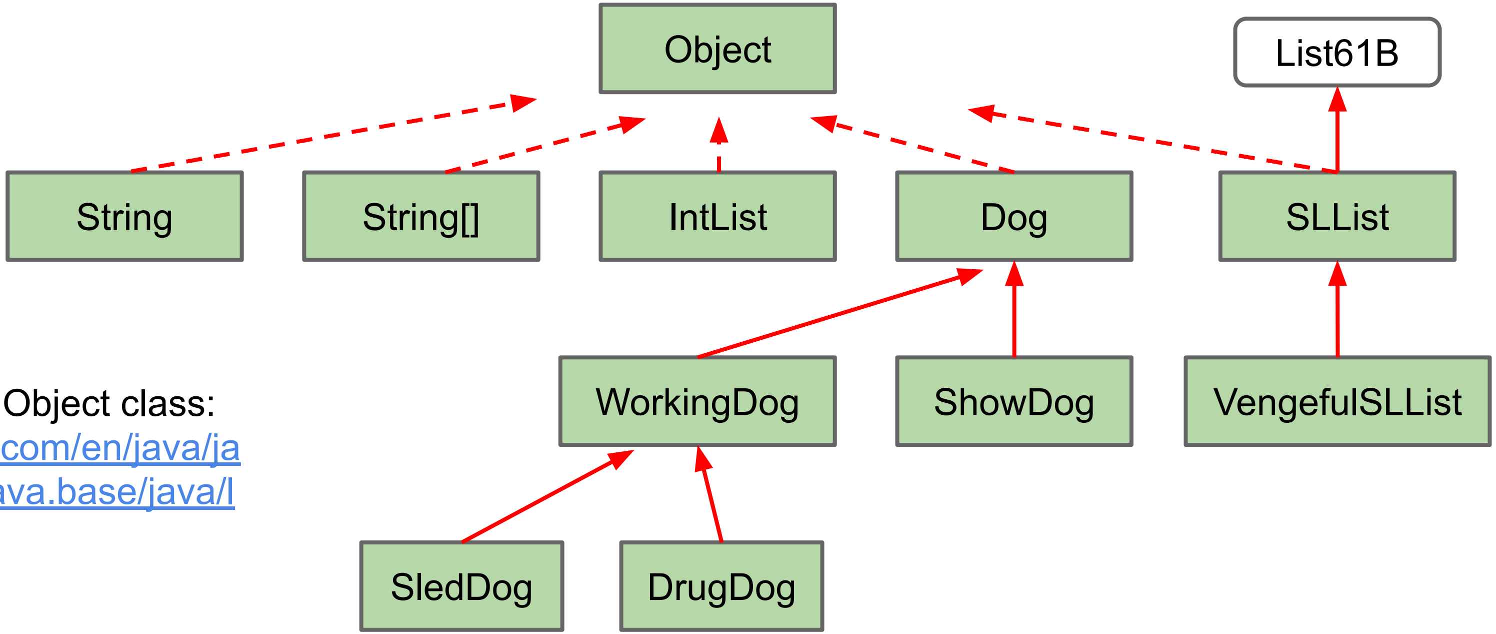
- Static and Dynamic Type
- Type Checking and Casting

Using Inheritance Safely

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: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html

Interfaces don't extend Object: http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.2

Object Methods

All classes are hyponyms of `Object`.

- `String toString()`
- `boolean equals(Object obj)`
- `int hashCode()`
- `Class<?>  getClass()`
- `protected Object clone()`
- `protected void finalize()`
- `void notify()`
- `void notifyAll()`
- `void wait()`
- `void wait(long timeout)`
- `void wait(long timeout, int nanos)`

Coming in another lecture soon.

Coming later.

Won’t discuss or use in 61B.

Thus every Java class has these methods. Amusingly `clone` is fundamentally broken.

Static and Dynamic Type

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

- **Static and Dynamic Type**
- Type Checking and Casting

Using Inheritance Safely

Question

Recall that if X is a superclass of Y, then an X variable can hold a reference to a Y.

Which print method do you think will run when the code below executes?

- SLList.removeLast()
- VengefulSLList.removeLast()

```java
public static void main(String[] args) {
   SLList<Integer> vsl = new VengefulSLList<Integer>(9);
   vsl.removeLast();
}
```

Question

Recall that if X is a superclass of Y, then an X variable can hold a reference to a Y.

Which print method do you think will run when the code below executes?

- SLList.removeLast()
- **VengefulSLList.removeLast().**
  And this is the sensible choice. But how does it work?
  - Before we can answer that, we need new terms: static and dynamic type.

```
public static void main(String[] args) {
   SLList<Integer> vsl = new VengefulSLList<Integer>(9);
   vsl.removeLast();
}
```

Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

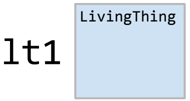
- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
    LivingThing lt1;
    lt1 = new Fox();
    Animal a1 = lt1;
    Fox h1 = new Fox();
    lt1 = new Squid();
}
```

Technically requires a “cast”. See next lecture.



	Static Type	Dynamic Type
lt1	LivingThing	null

Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

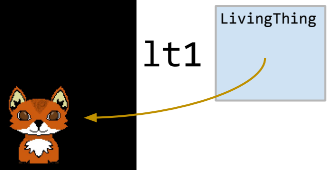
- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
   LivingThing lt1;
   lt1 = new Fox();
   Animal a1 = lt1;
   Fox h1 = new Fox();
   lt1 = new Squid();
}
```

Technically requires a “cast”. See next lecture.



	Static Type	Dynamic Type
lt1	LivingThing	Fox

Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

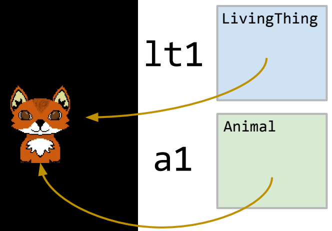
- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
    LivingThing lt1;
    lt1 = new Fox();
  → Animal a1 = lt1;
    Fox h1 = new Fox();
    lt1 = new Squid();
}
```

Technically requires a “cast”. See next lecture.



	Static Type	Dynamic Type
lt1	LivingThing	Fox
a1	Animal	Fox

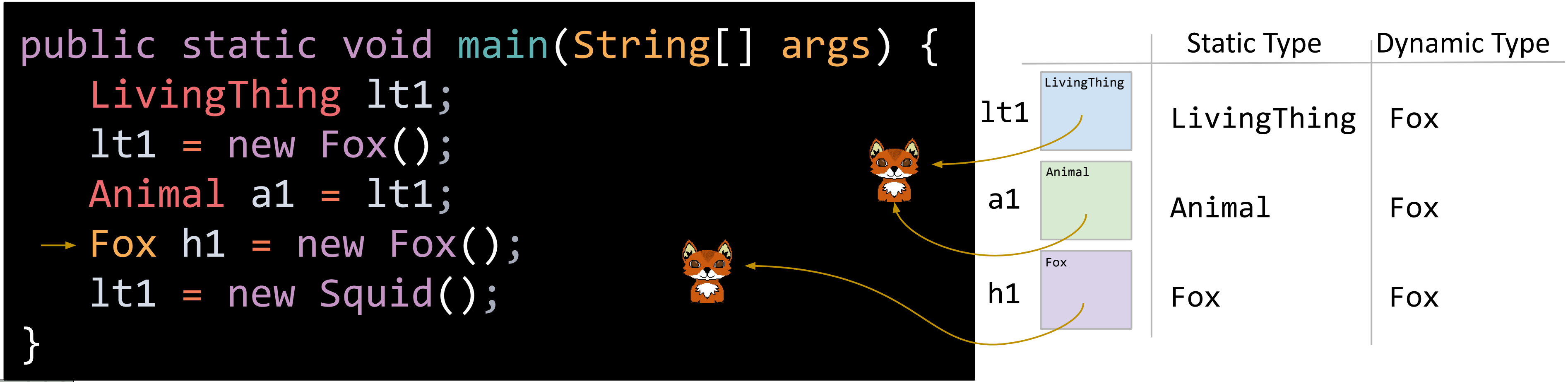
Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.



```
public static void main(String[] args) {
    LivingThing lt1;
    lt1 = new Fox();
    Animal a1 = lt1;
→   Fox h1 = new Fox();
    lt1 = new Squid();
}
```

	Static Type	Dynamic Type
lt1	LivingThing	Fox
a1	Animal	Fox
h1	Fox	Fox

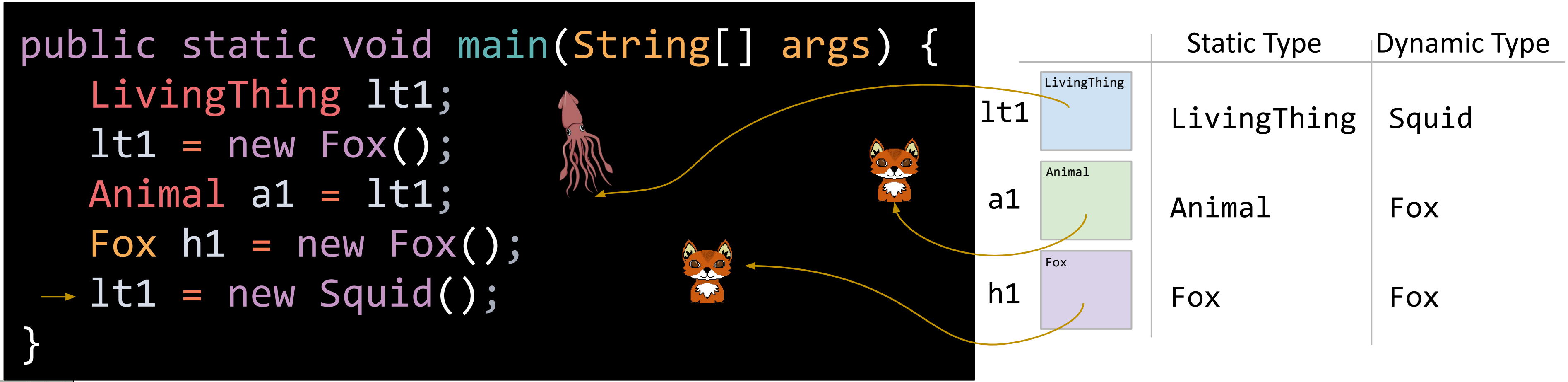
Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.



```
public static void main(String[] args) {
    LivingThing lt1;
    lt1 = new Fox();
    Animal a1 = lt1;
    Fox h1 = new Fox();
 →  lt1 = new Squid();
}
```

	Static Type	Dynamic Type
lt1	LivingThing	Squid
a1	Animal	Fox
h1	Fox	Fox

Dynamic Method Selection For Overridden Methods

Suppose we call a method of an object using a variable with:

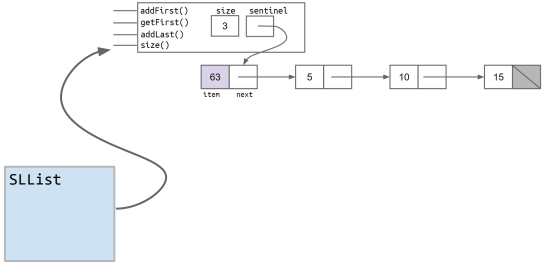
- compile-time type X
- run-time type Y

Then if Y **overrides** the method, Y’s method is used instead.

- This is known as “dynamic method selection”.

This term is a bit obscure.

```
public static void main(String[] args) {
   LivingThing lt1;
   lt1 = new Fox();
   Animal a1 = lt1;
   Fox h1 = new Fox();
   lt1 = new Squid();
}
```



Static Type
SLList

Dynamic Type
VengefulSLList

Older Versions of 61B (pre-2023)

In older versions of this class, the section on Dynamic Method Selection included a tricky corner case where a subclass overloads (rather than overrides) a superclass method.

- Even older versions went even deeper, showing what happens when subclasses have variables with the same name as their superclass.

Students spent a great deal of time on something that isn't ultimately very important. This is not a class about Java minutiae, so I cut this material.

- Example, the infamous Bird/Falcon/gulgate problem from Spring 2017: https://hkn.eecs.berkeley.edu/examfiles/cs61b_sp17_mt1.pdf
- If you are doing problems where the behavior of the DMS is highly counterintuitive, it is probably out of scope.
- See these extra slides or bonus video A, then bonus video B if you're curious.

Type Checking and Casting

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

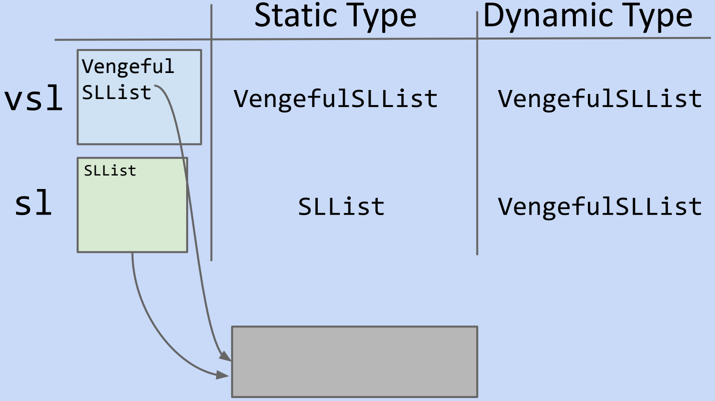
- Static and Dynamic Type
- **Type Checking and Casting**

Using Inheritance Safely

Dynamic Method Selection and Type Checking Puzzle

For each line of code, determine:

- Does that line cause a compilation error?
- Which method does dynamic method selection use?



	Static Type	Dynamic Type
vsl	VengefulSLList	VengefulSLList
sl	SLList	VengefulSLList

```
public static void main(String[] args) {
   VengefulSLList<Integer> vsl =
           new VengefulSLList<Integer>(9);
   SLList<Integer> sl = vsl;

   sl.addLast(50);
   sl.removeLast();

   sl.printLostItems();
   VengefulSLList<Integer> vsl2 = sl;
}
```

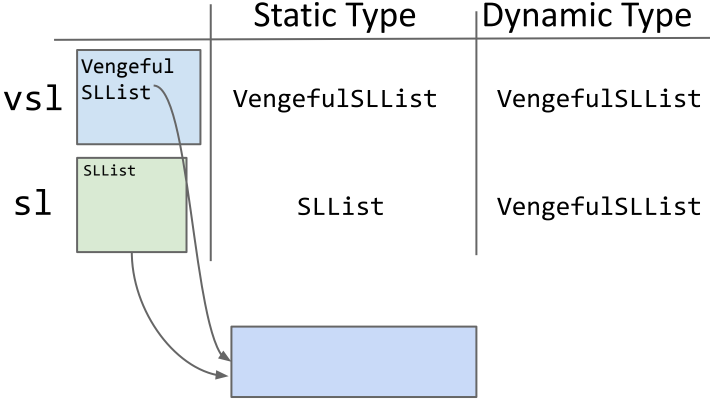
Reminder: VengefulSLList overrides removeLast and provides a new method called printLostItems.

Reminder: Dynamic Method Selection

Also called dynamic type.

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

- sl’s runtime type: VengefulSLList.



	Static Type	Dynamic Type
vsl	VengefulSLList	VengefulSLList
sl	SLList	VengefulSLList

```
public static void main(String[] args) {
   VengefulSLList<Integer> vsl =
            new VengefulSLList<Integer>(9);
   SLList<Integer> sl = vsl;

   sl.addLast(50);
   sl.removeLast();

   sl.printLostItems();
   VengefulSLList<Integer> vsl2 = sl;
}
```

VengefulSLList doesn’t override, uses SLList’s.

Uses VengefulSLList’s.

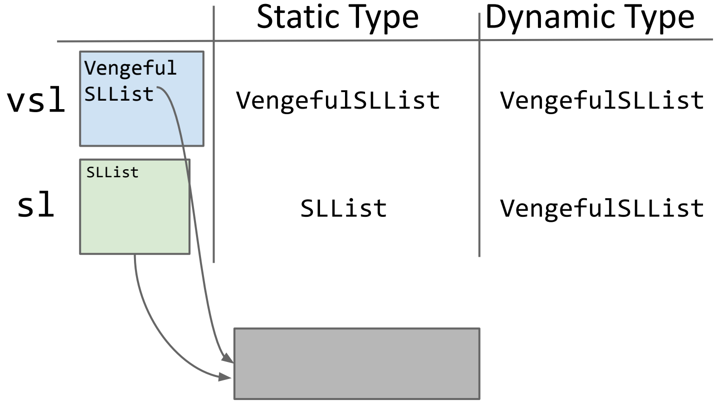
Reminder: VengefulSLList overrides removeLast and provides a new method called printLostItems.

Compile-Time Type Checking

Also called static type.

Compiler allows method calls based on **compile-time** type of variable.

- sl's runtime type: VengefulSLList.
- But cannot call printLostItems.



	Static Type	Dynamic Type
vsl	VengefulSLList	VengefulSLList
sl	SLList	VengefulSLList

```
public static void main(String[] args) {
   VengefulSLList<Integer> vsl =
            new VengefulSLList<Integer>(9);
   SLList<Integer> sl = vsl;

   sl.addLast(50);
   sl.removeLast();

   sl.printLostItems();
   VengefulSLList<Integer> vsl2 = sl;
}
```

Compilation error!

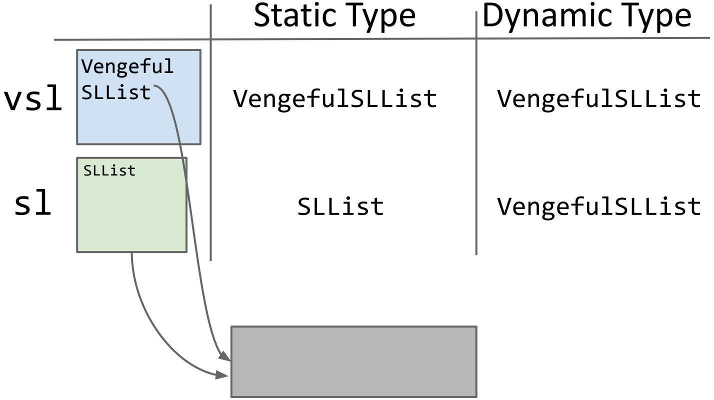
Reminder: VengefulSLList overrides removeLast and provides a new method called printLostItems.

Compile-Time Type Checking

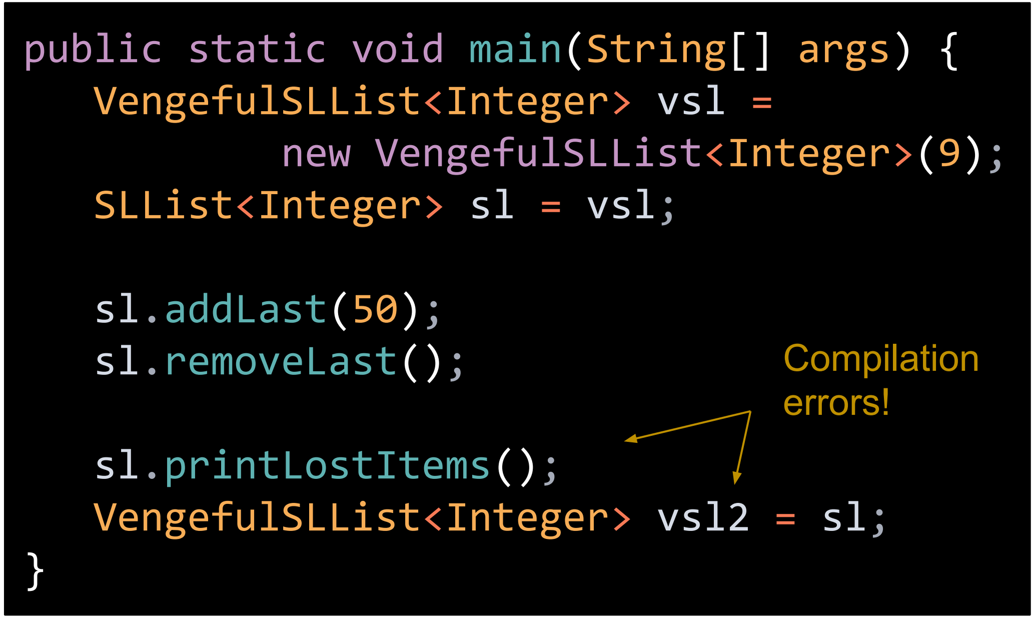
Also called static type.

Compiler allows method calls based on **compile-time** type of variable.

- sl's runtime type: VengefulSLList.
- But cannot call printLostItems.



	Static Type	Dynamic Type
vsl	VengefulSLList	VengefulSLList
sl	SLList	VengefulSLList



```
public static void main(String[] args) {
    VengefulSLList<Integer> vsl =
            new VengefulSLList<Integer>(9);
    SLList<Integer> sl = vsl;

    sl.addLast(50);
    sl.removeLast();

    sl.printLostItems();
    VengefulSLList<Integer> vsl2 = sl;
}
```

Compilation errors!

Compiler also allows assignments based on compile-time types.

- Even though sl's runtime-type is VengefulSLList, cannot assign to vsl2.
- 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.

```java
SLList<Integer> sl = new VengefulSLList<Integer>();
```

- Compile-time type of right hand side (RHS) expression is VengefulSLList.
- A VengefulSLList is-an SLList, so assignment is allowed.

```java
VengefulSLList<Integer> vsl = new SLList<Integer>();
```

Compilation error!

- Compile-time type of RHS expression is SLList.
- An SLList is not necessarily a VengefulSLList, so compilation error results.

Compile-Time Types and Expressions

Expressions have compile-time types:

- Method calls 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 specifying the compile-time type of any expression.

- Put desired type in parenthesis before the expression.
- Examples:
  - Compile-time type Dog:

```
maxDog(frank, frankJr);
```

  - Compile-time type Poodle:

```
(Poodle) maxDog(frank, frankJr);
```

Tells compiler to pretend it sees a particular type.

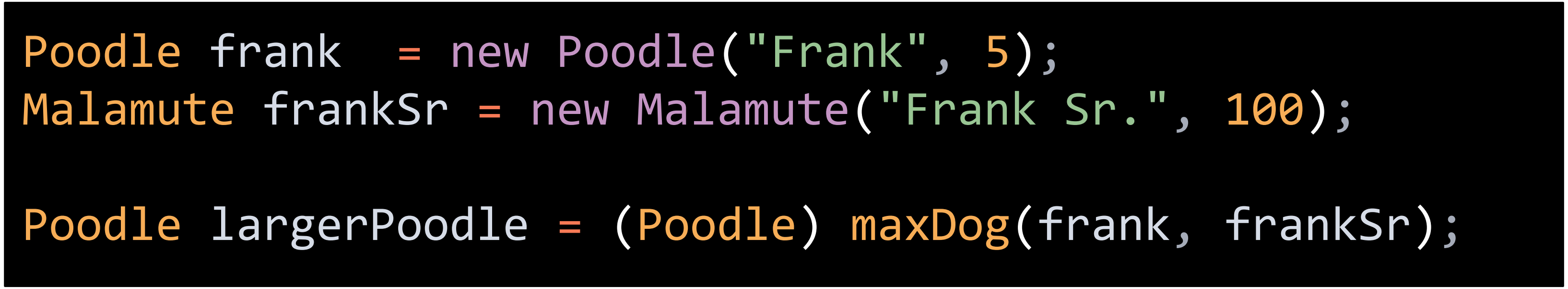
```
Poodle frank  = new Poodle("Frank", 5);
Poodle frankJr = new Poodle("Frank Jr.", 15);

Dog largerDog = maxDog(frank, frankJr);
Poodle largerPoodle = (Poodle) maxDog(frank, frankJr);
```

Compilation OK!
RHS has compile-time type Poodle.

Casting

Casting is a powerful but dangerous tool.

- Tells Java to treat an expression as having a different compile-time type.
- In example below, effectively tells the compiler to ignore its type checking duties.
- Does not actually change anything: sunglasses don’t make the world dark.



```
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.

- So much for .class files being verifiably type checked...

Using Inheritance Safely

Lecture 8, CS61B, Fall 2024

Interface Inheritance

- The Desire for Generality
- Hypernyms and Hyponyms
- Interface and Implements Keywords
- Overriding vs. Overloading
- Interface Inheritance

Implementation Inheritance

- Extends Keyword: Rotating SLList
- Super Keyword: Vengeful SLList
- A Boring Constructor Gotcha
- The Object Class

Calling Overridden Methods

- Static and Dynamic Type
- Type Checking and Casting

Using Inheritance Safely

Interface vs. Implementation Inheritance

Interface Inheritance (a.k.a. what):

- Allows you to generalize code in a powerful, simple way.

Implementation Inheritance (a.k.a. how):

- Allows code-reuse: Subclasses can rely on superclasses or interfaces.
  - Example: addFirst() implemented in SLList.java, but used in RotatingSLList.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.

The Dangers of Implementation Inheritance

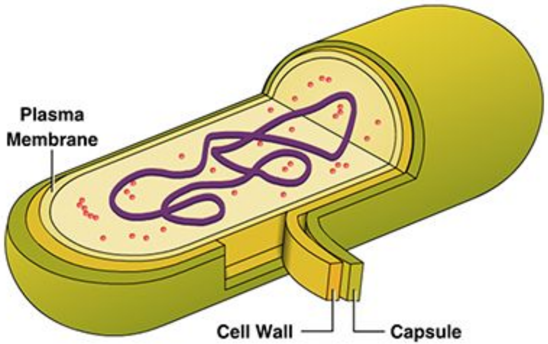
Particular Dangers of Implementation Inheritance

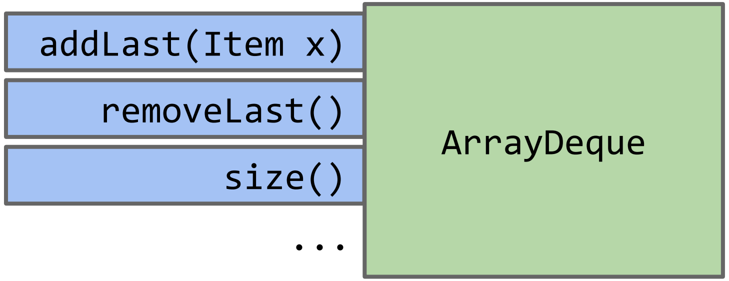
- Makes it harder to keep track of where something was actually implemented (though a good IDE makes this better).
- Rules for resolving conflicts can be arcane. Won't cover in 61B.
  - Example: What if a method is both overloaded and overridden?
- Encourages overly complex code (especially with novices).
  - Common mistake: Has-a vs. Is-a!
- Breaks encapsulation!

Modules and Encapsulation [Shewchuk]

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.

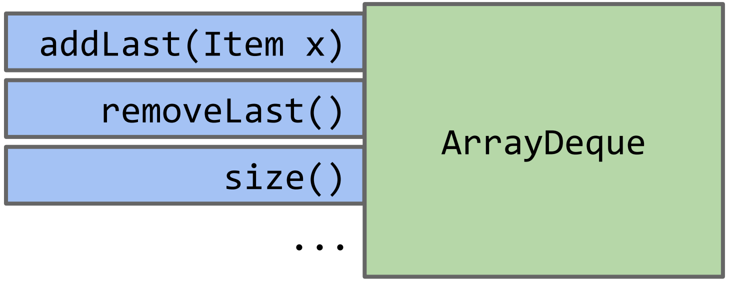




Abstraction Barriers

As the user of an ArrayDeque, you cannot observe its internals.

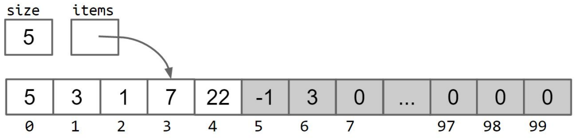
- Even when writing tests, you don't (usually) want to peer inside.



Java is a great language for enforcing abstraction barriers with syntax.

```
{5, 3, 1, 7, 22}
```

Implementation



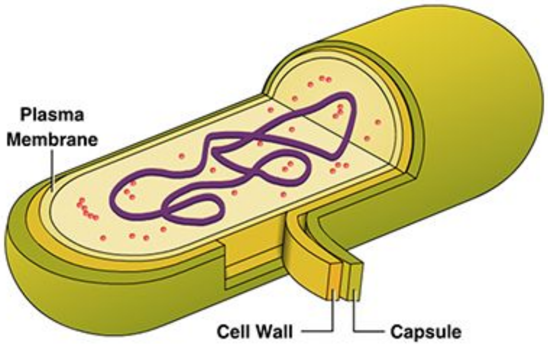
5	3	1	7	22	-1	3	0	...	0	0	0
0	1	2	3	4	5	6	7		97	98	99

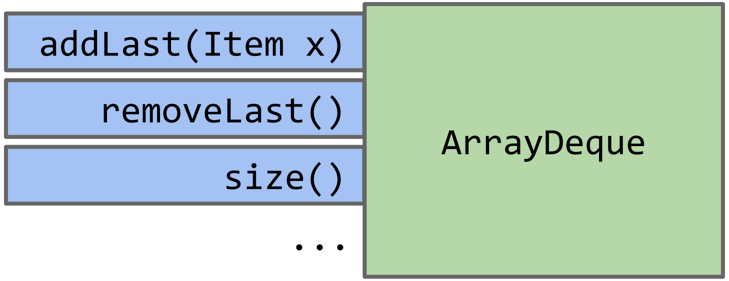
Modules and Encapsulation [Shewchuk]

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 variables private. Methods like `resize` private.
- Implementation inheritance (e.g. extends) breaks encapsulation!
  - (Optional) To see why, check out this video and this video.
  - Intuition: A subclass can "see" the implementation of its superclass.





addLast(Item x)

removeLast()

size()

...

ArrayDeque