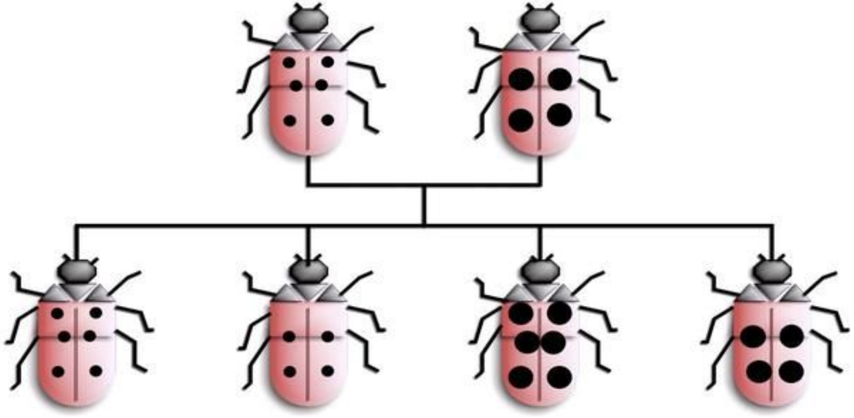


Lecture 9 (Inheritance 2)

Extends, Casting, Higher Order Functions

CS61B, Spring 2024 @ UC Berkeley

Slides credit: Josh Hug

Rotating SLList

Lecture 9, CS61B, Spring 2024

The Extends Keyword

- **Rotating SLList**
- Vengeful SLList
- A Boring Constructor Gotcha

Implementation Inheritance

- The Object Class
- Is-A vs. Has-A, java.util.Stack
- Encapsulation
- Implementation Inheritance Breaks Encapsulation

Type Checking and Casting

Higher Order Functions in Java

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

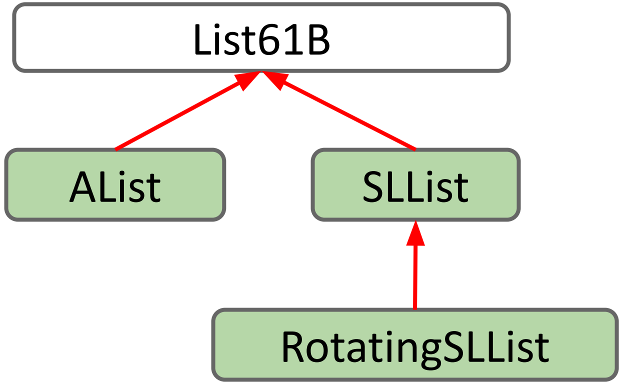
If you want one class to be a hyponym of another *class*, you use **extends.**

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

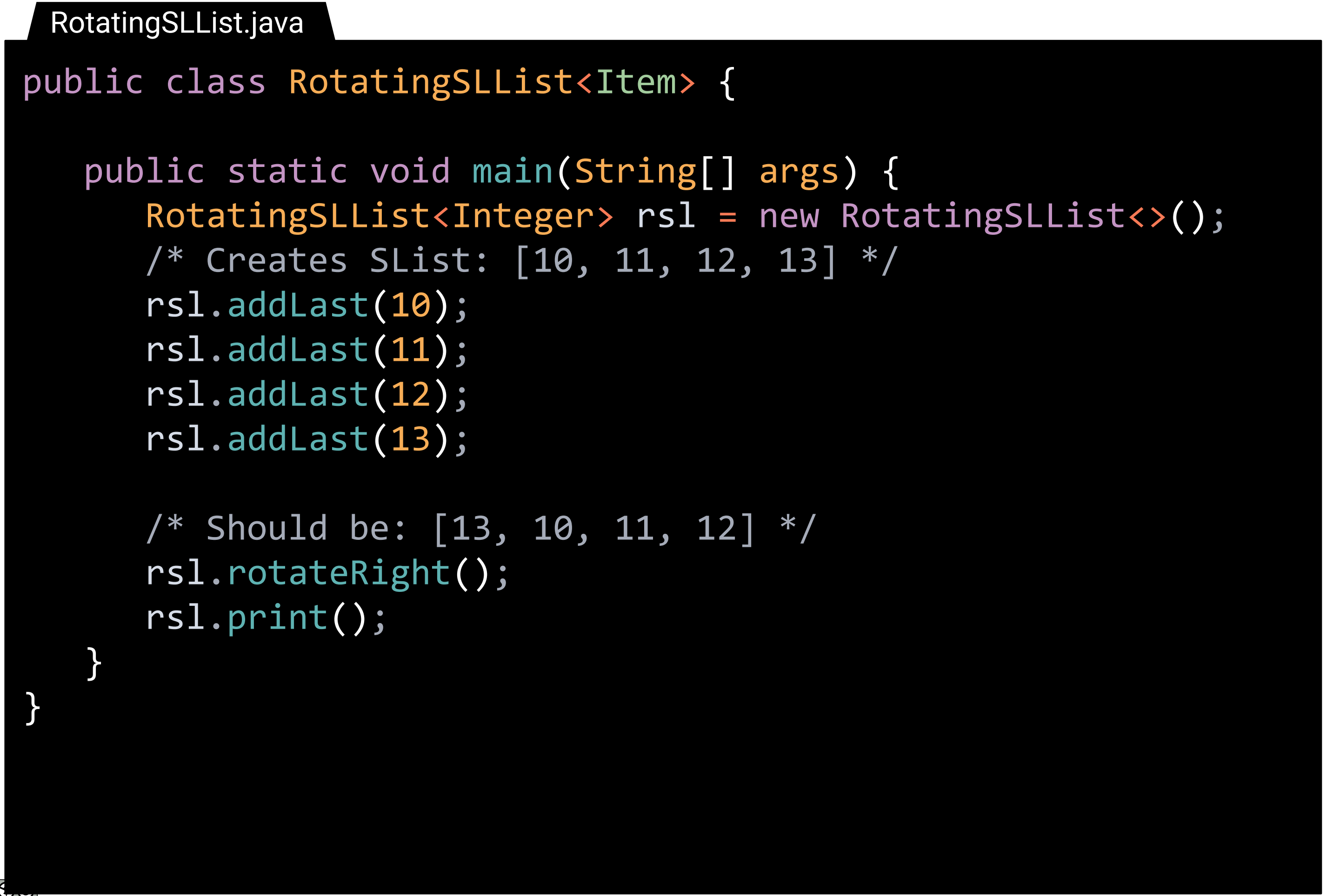
- rotateRight(): Moves back item the front.

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

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



Demo: Rotating SLList

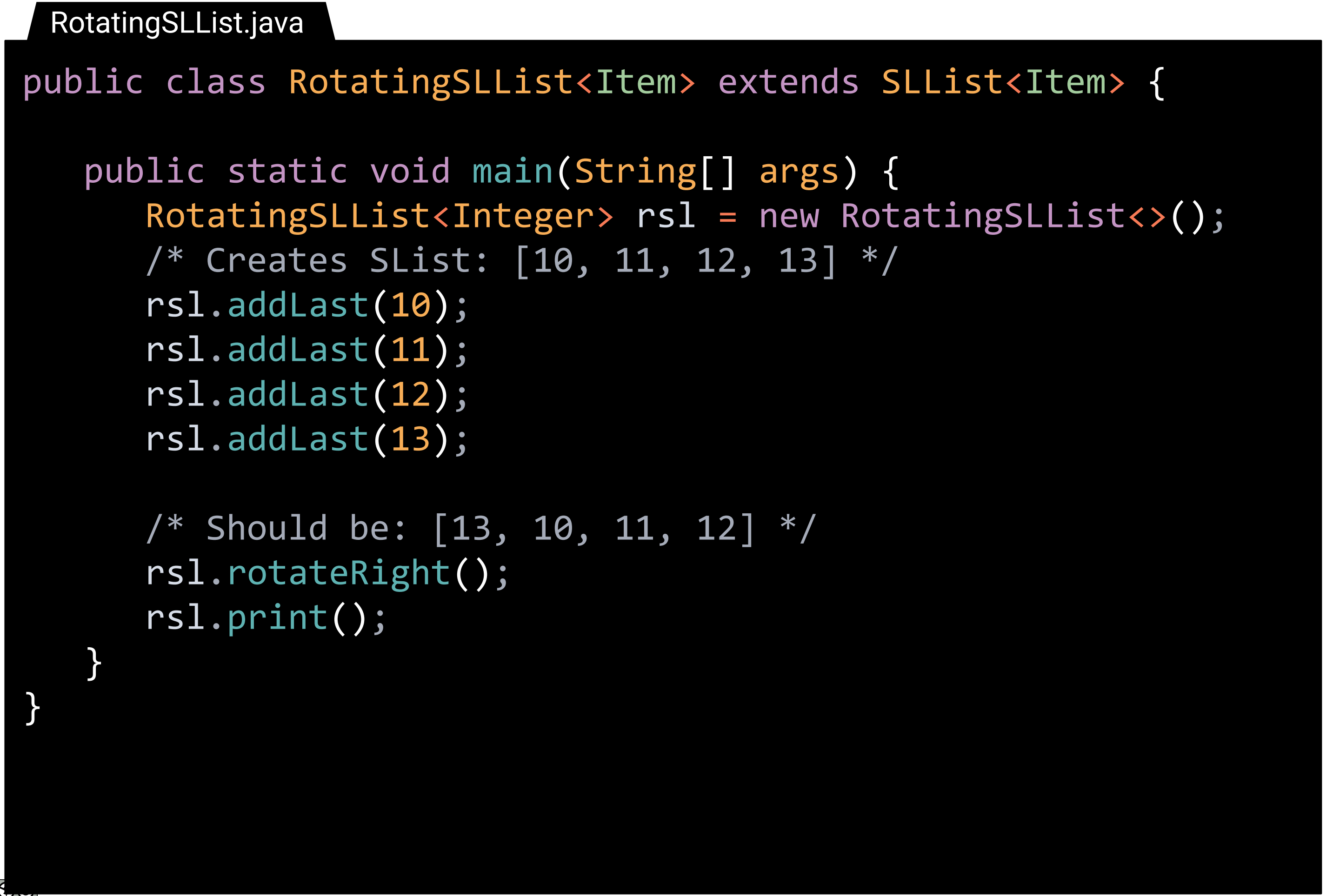


RotatingSLList.java

```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.

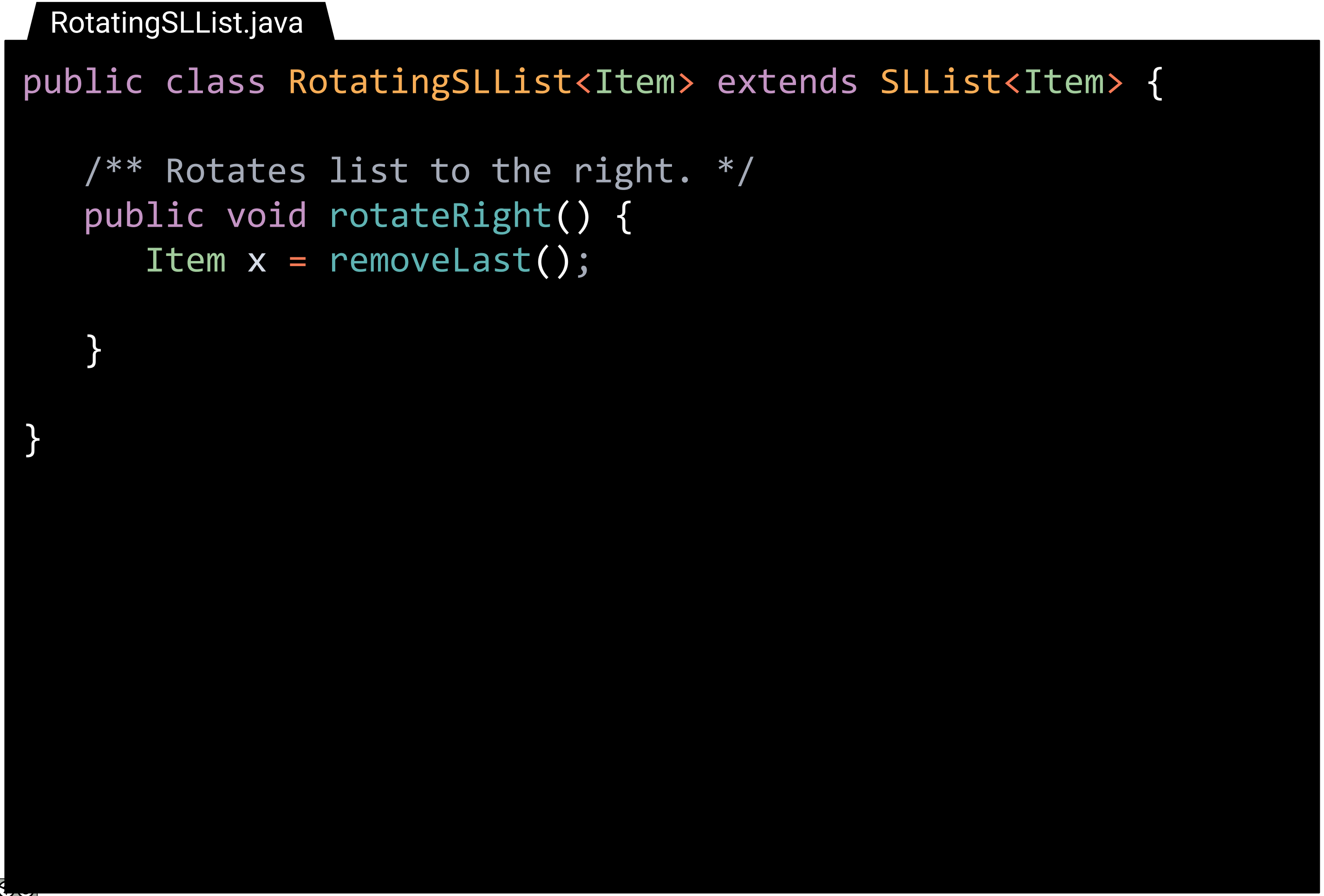
The rotateRight method is still missing.

Demo: Rotating SLList

RotatingSLList.java

```
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.

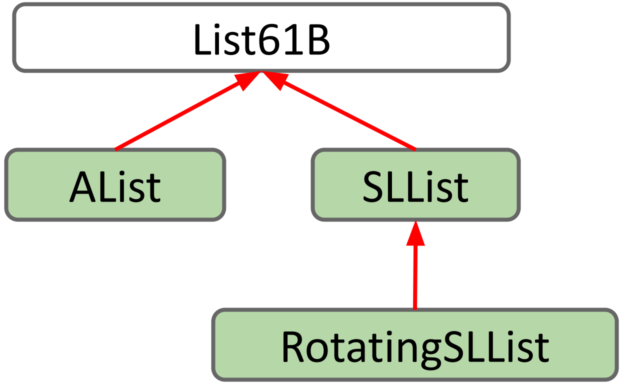
Constructors are not inherited.

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.



Vengeful SLList

Lecture 9, CS61B, Spring 2024

The Extends Keyword

- Rotating SLList
- **Vengeful SLList**
- A Boring Constructor Gotcha

Implementation Inheritance

- The Object Class
- Is-A vs. Has-A, java.util.Stack
- Encapsulation
- Implementation Inheritance Breaks Encapsulation

Type Checking and Casting

Higher Order Functions in Java

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();
   }
}
```

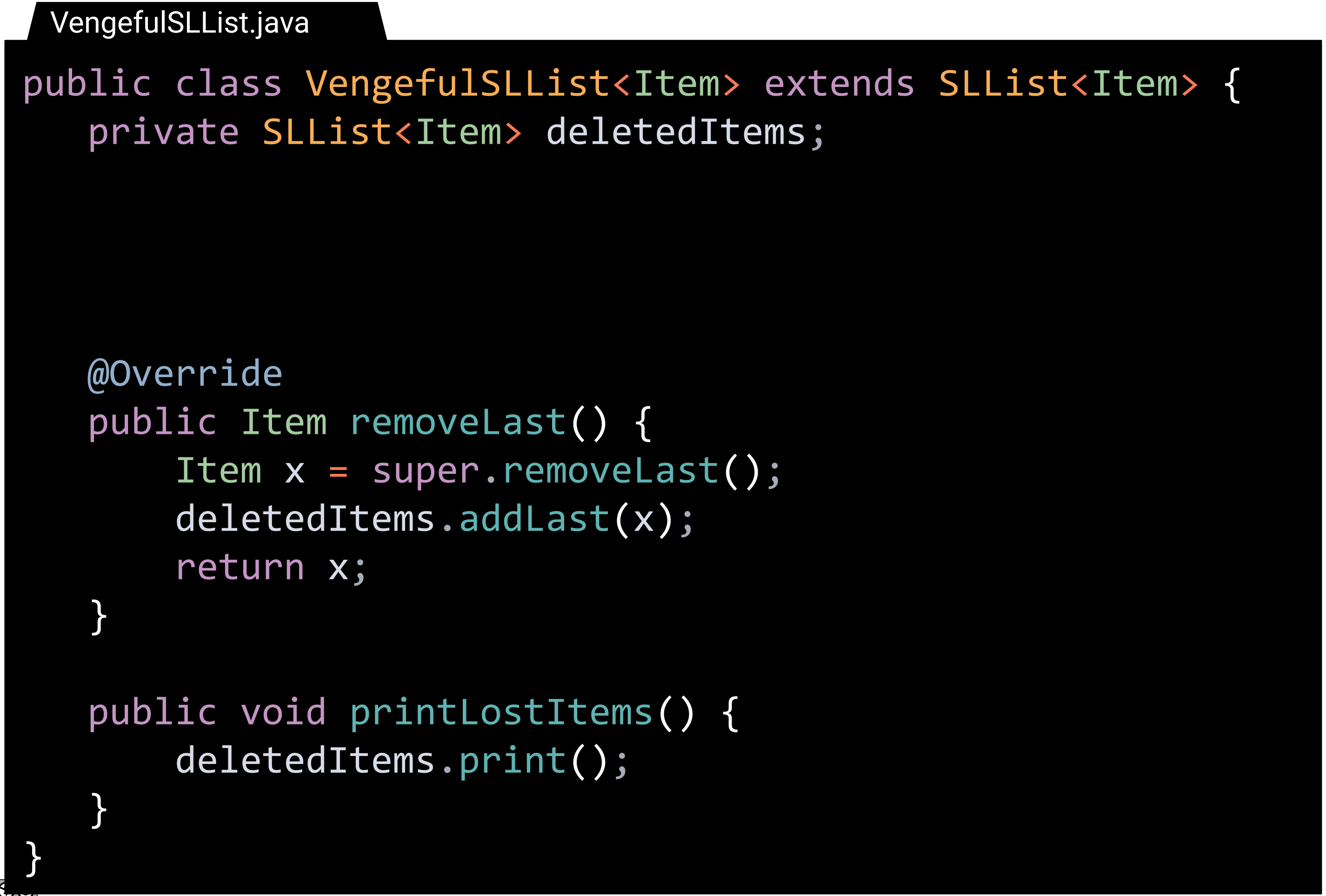
Solution: Use the super keyword to call SLList's removeLast method.

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

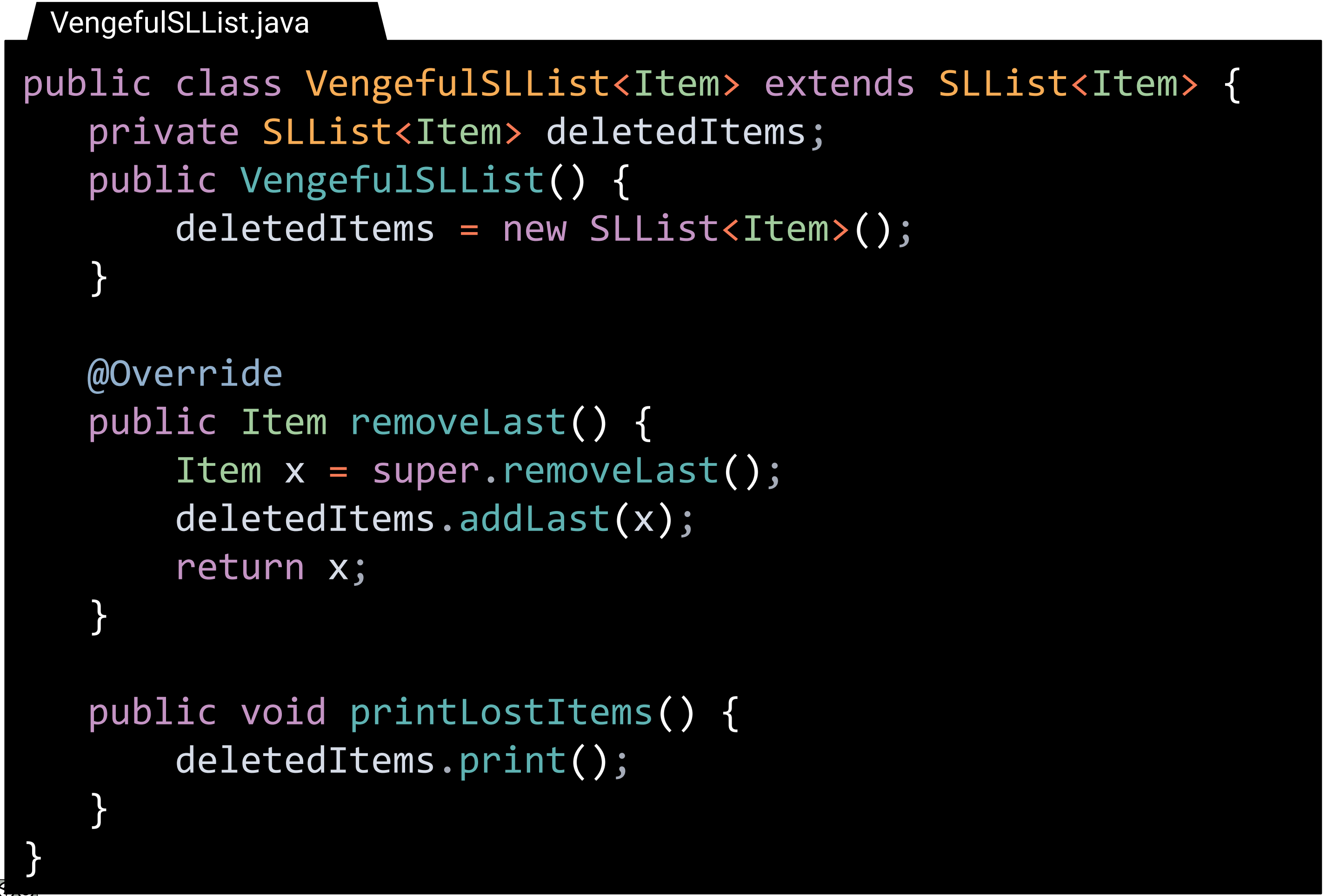
VengefulSLList.java

```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 9, CS61B, Spring 2024

The Extends Keyword

- Rotating SLList
- Vengeful SLList
- **A Boring Constructor Gotcha**

Implementation Inheritance

- The Object Class
- Is-A vs. Has-A, java.util.Stack
- Encapsulation
- Implementation Inheritance Breaks Encapsulation

Type Checking and Casting

Higher Order Functions in Java

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

```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

```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.

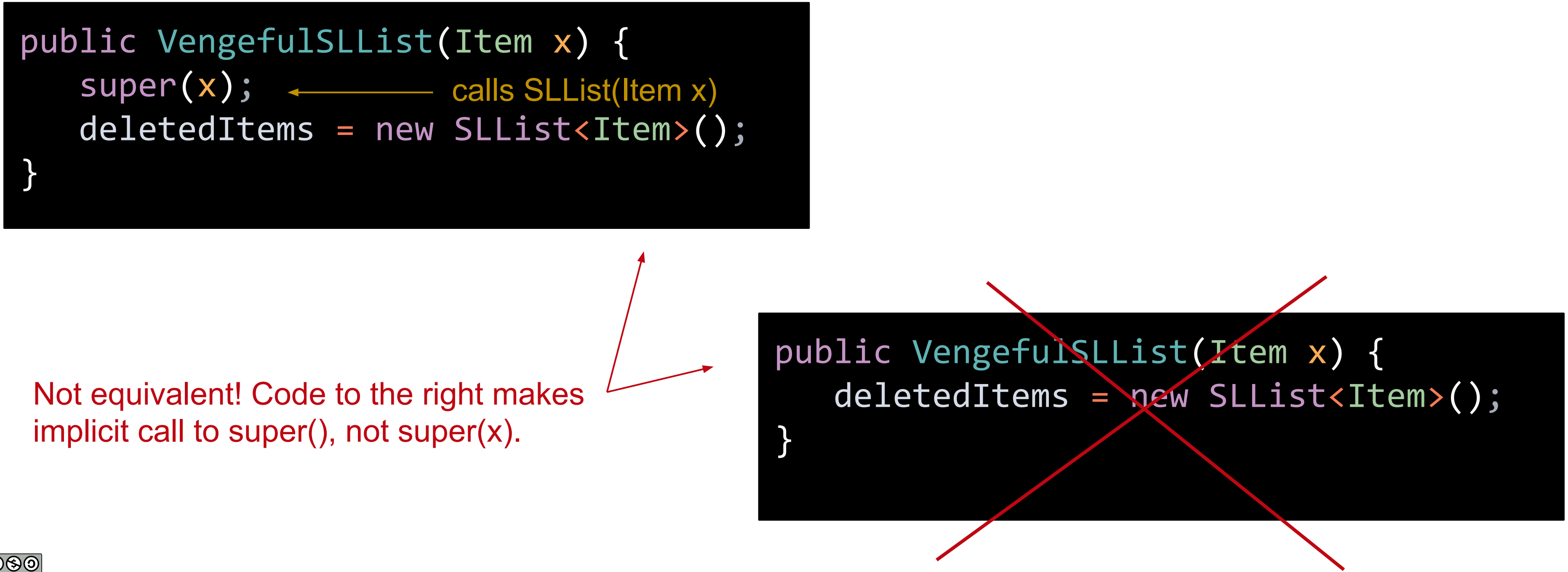
```
public VengefulSLList() {
    deletedItems = new SLList<Item>();
}
```

```
public VengefulSLList() {
    super();  ←  must come first!
    deletedItems = new SLList<Item>();
}
```

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>();
}
```

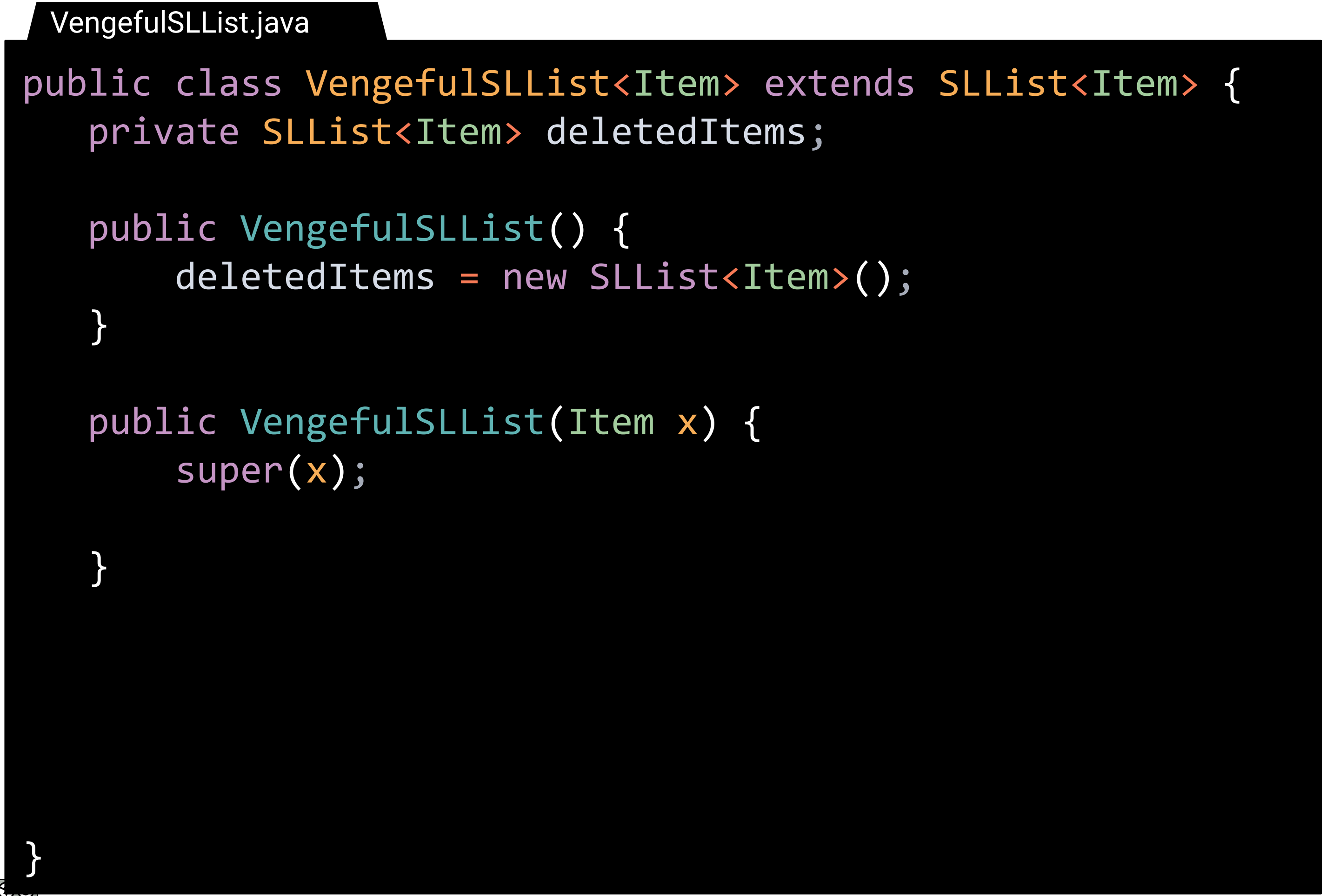
Coding Demo: Vengeful SLList

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

```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

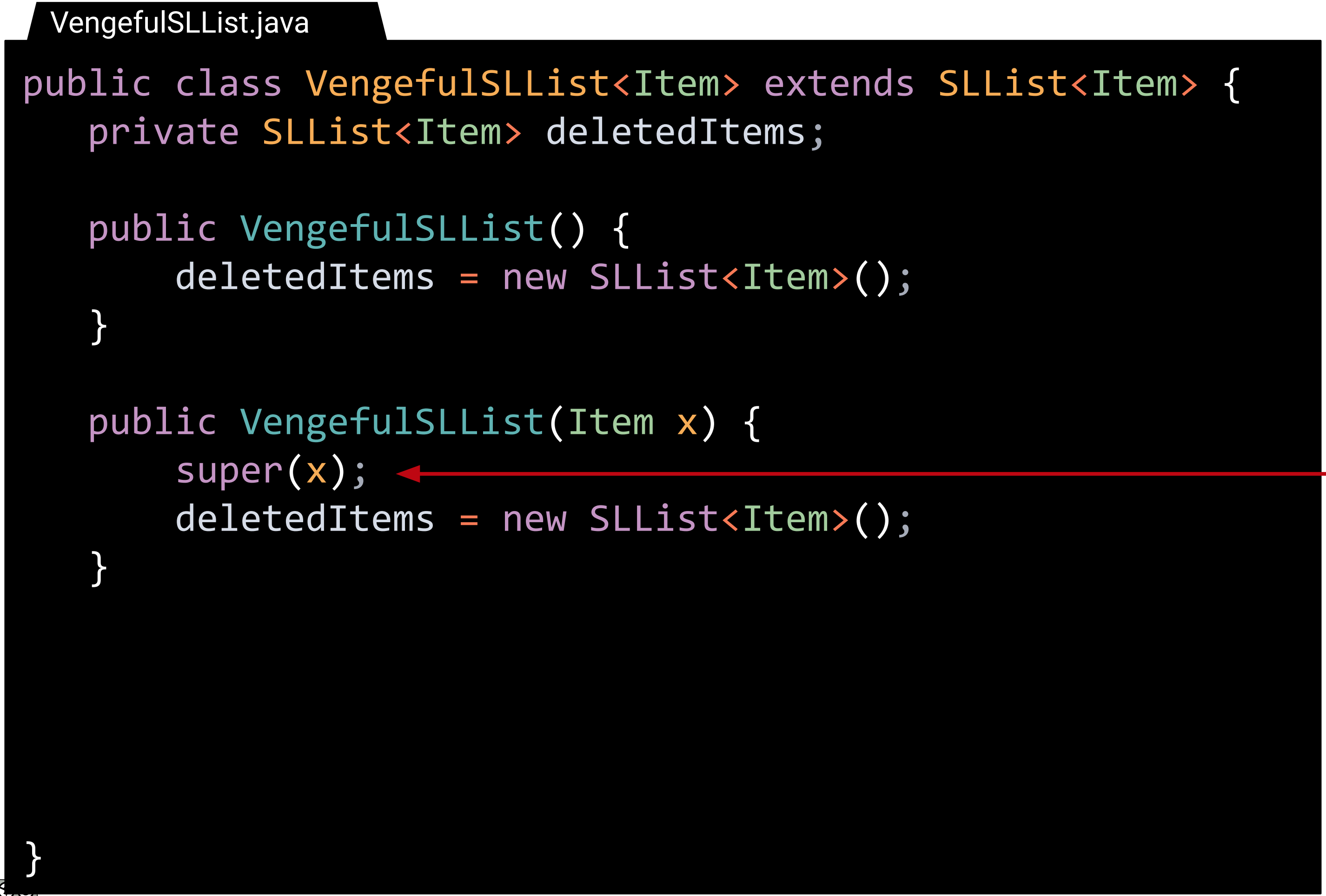
VengefulSLList.java

```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



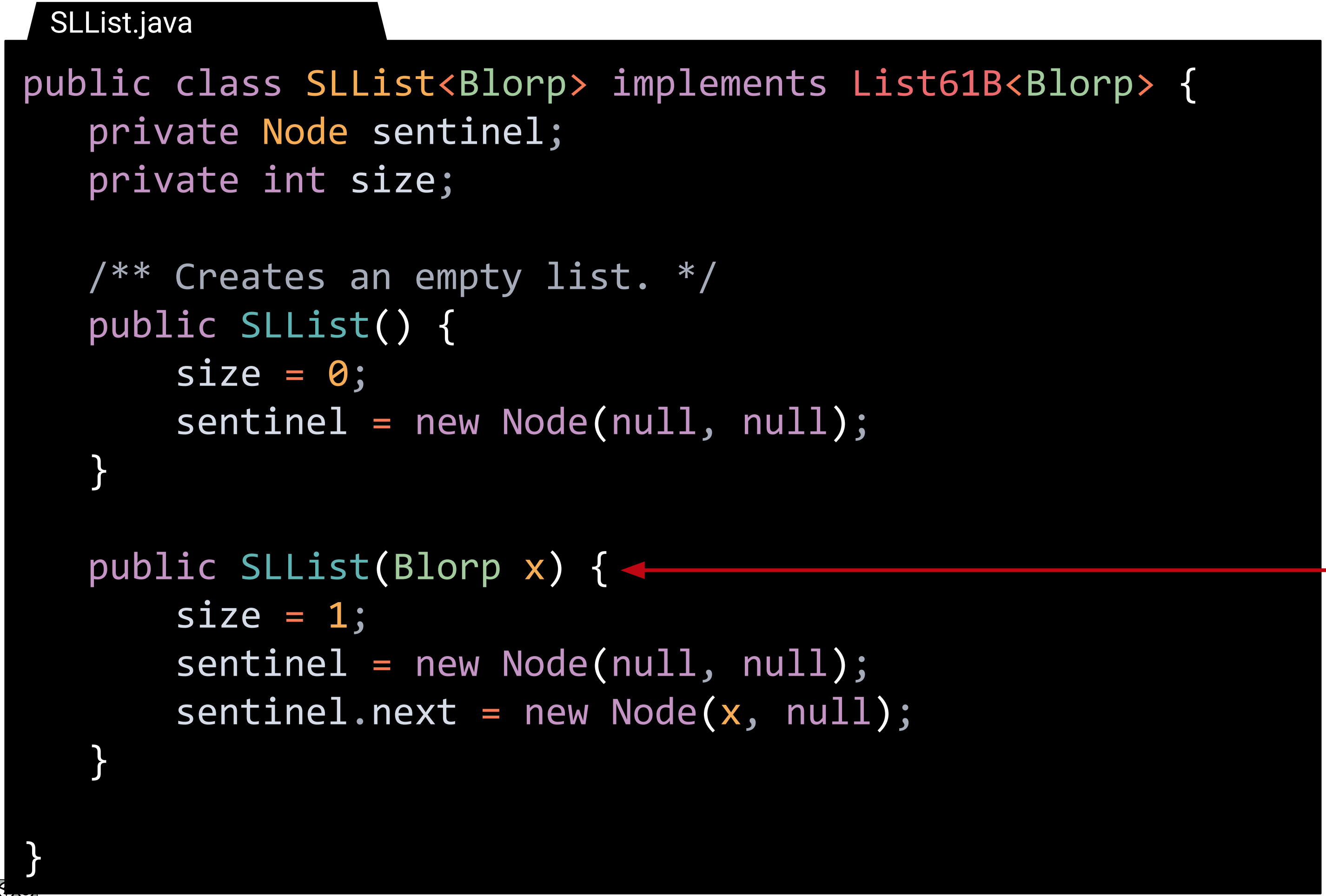
VengefulSLList.java

```
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

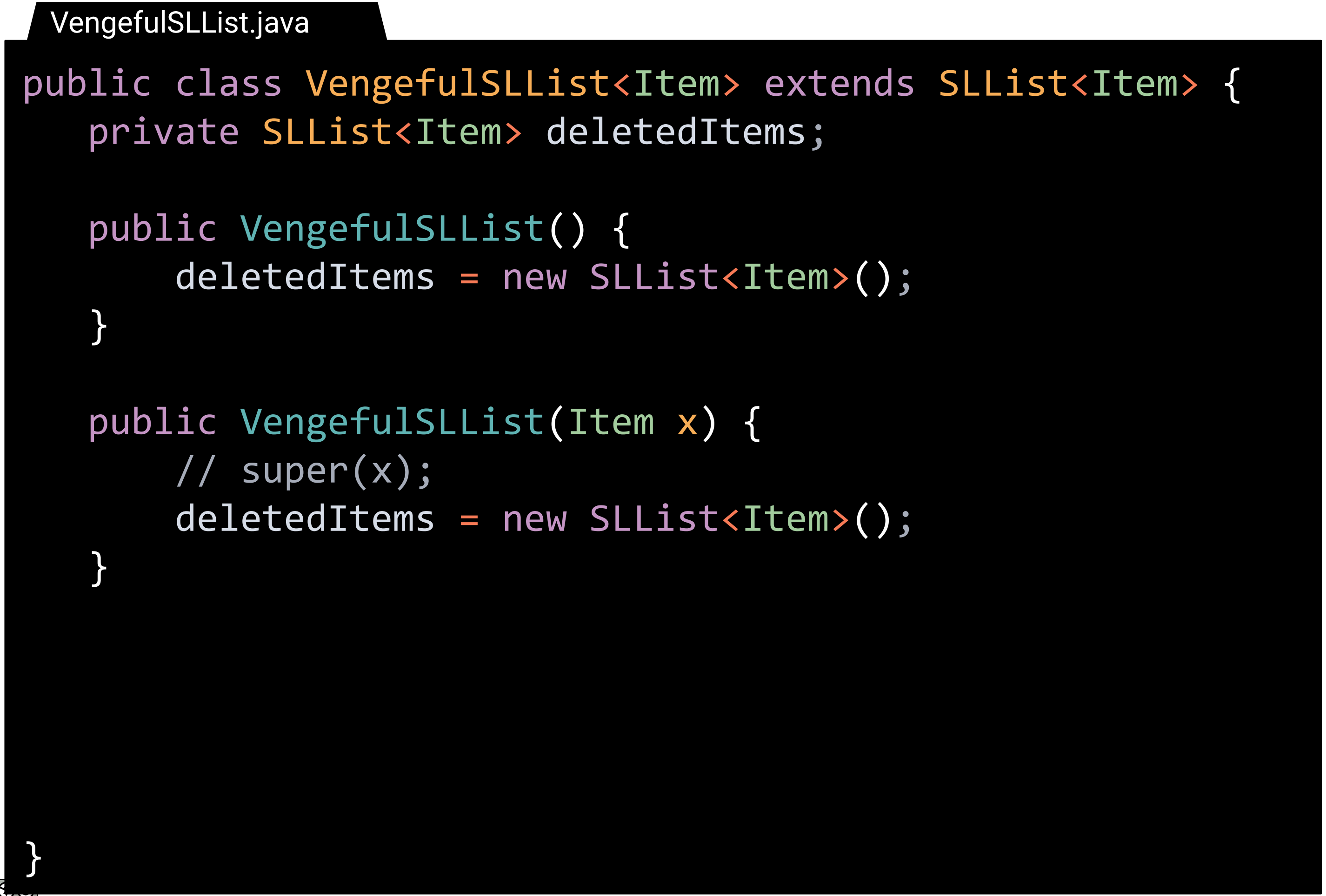


SLList.java

```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

```
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 9, CS61B, Spring 2024

The Extends Keyword

- Rotating SLList
- Vengeful SLList
- A Boring Constructor Gotcha

Implementation Inheritance

- **The Object Class**
- Is-A vs. Has-A, java.util.Stack
- Encapsulation
- Implementation Inheritance Breaks Encapsulation

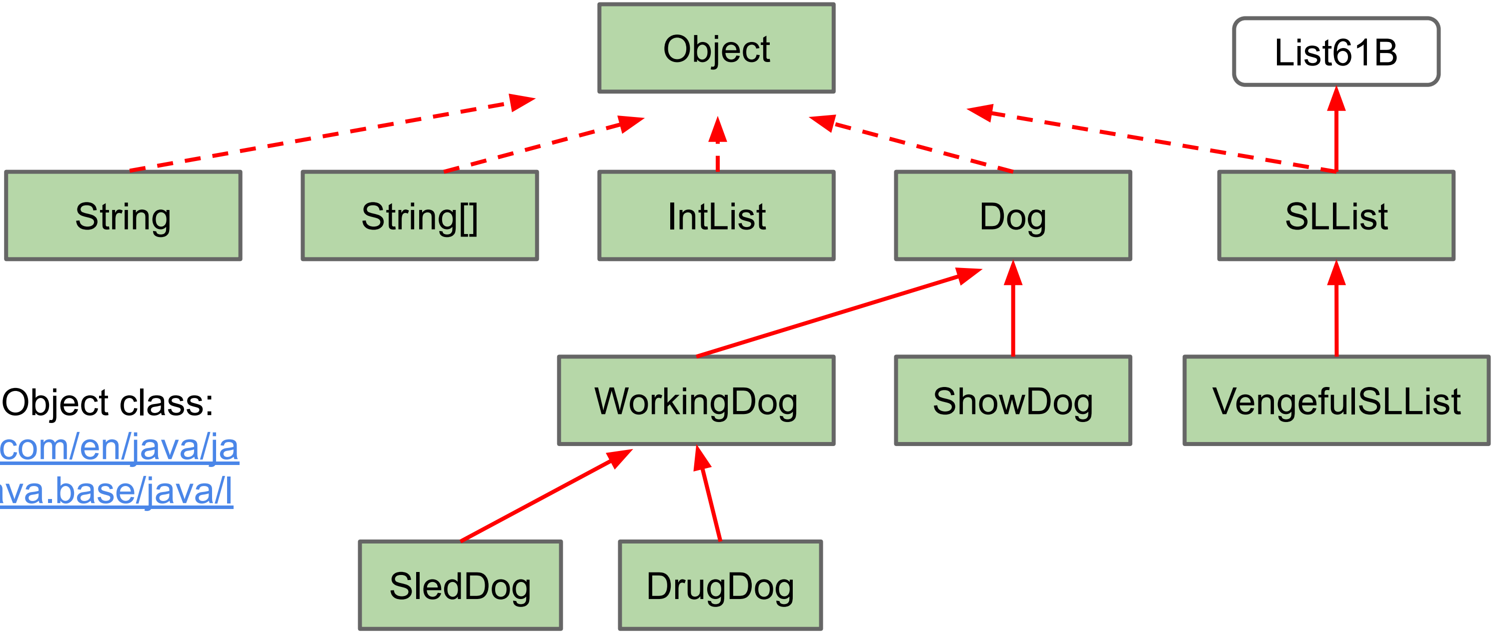
Type Checking and Casting

Higher Order Functions in Java

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.

Is-A vs. Has-A, java.util.Stack

Lecture 9, CS61B, Spring 2024

The Extends Keyword

- Rotating SLList
- Vengeful SLList
- A Boring Constructor Gotcha

Implementation Inheritance

- The Object Class
- **Is-A vs. Has-A, java.util.Stack**
- Encapsulation
- Implementation Inheritance Breaks Encapsulation

Type Checking and Casting

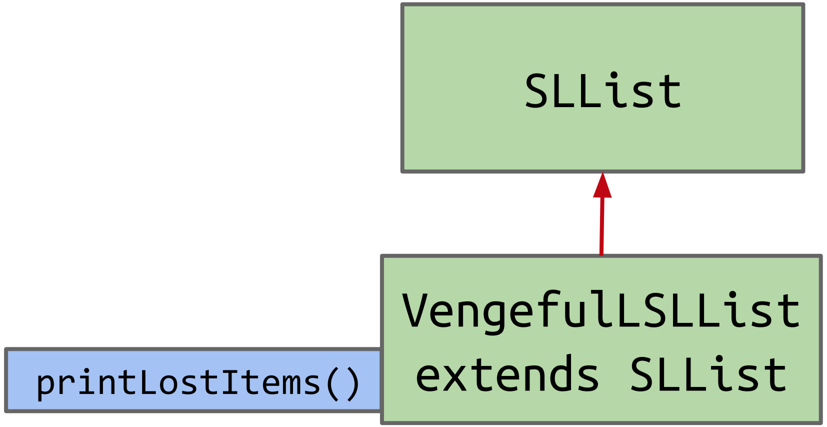
Higher Order Functions in Java

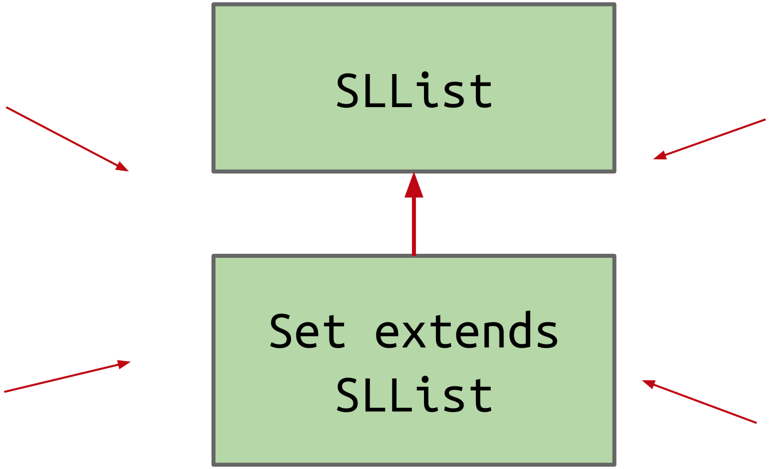
Is-a vs. Has-A

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.





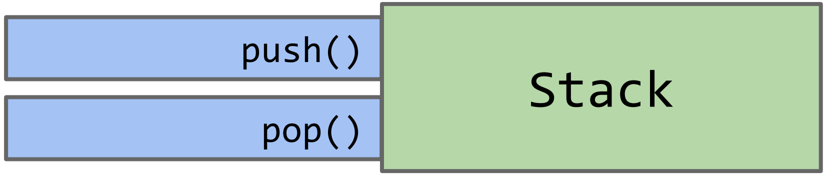
This is an abomination.

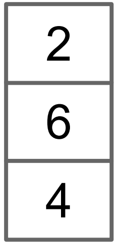
Example: Stack

The Stack abstract data type (ADT) supports the following operations:

- `push(x)`: Puts x on top of the stack.
- `pop()`: Removes and returns the top item from the stack.

The Java designers made a grave error when they wrote `java.util.Stack`.

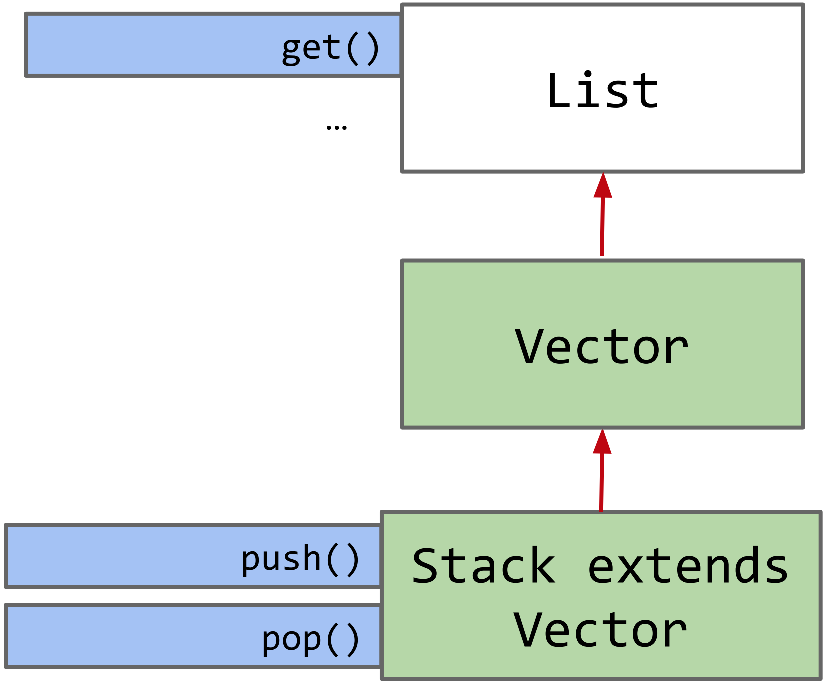




Is-a vs. Has-A

Example of a Has-A error in Java: The Stack class.

- They decided that Stack extends Vector (which implements List).
- Thus Stacks have all list operations.



A Vector is a slightly different version of an ArrayList.

Stack (if it had been done correctly using has-a)

Stacks are supposed to be simple:

- push
- pop
- size

Could have been implemented simply:

- Each Stack “has-a” LinkedList that stores its data.

```
public class Stack<T> {
   private LinkedList<T> items = new LinkedList<>();

   public void push(T x) { items.addLast(x); }
   public T pop() { return items.removeLast(); }
   public int size() { return items.size();}
}
```

Stack (because it is-a Vector)

But java.util.Stack is:

- push
- pop
- add
- contains
- elements
- ensureCapacity
- firstElement
- get
- indexOf
- insertElementAt
- lastElement
- lastIndexOf
- remove
- removeRange
- …

Encapsulation

Lecture 9, CS61B, Spring 2024

The Extends Keyword

- Rotating SLList
- Vengeful SLList
- A Boring Constructor Gotcha

Implementation Inheritance

- The Object Class
- Is-A vs. Has-A, java.util.Stack
- **Encapsulation**
- Implementation Inheritance Breaks Encapsulation

Type Checking and Casting

Higher Order Functions in Java

Complexity: The Enemy

When building large programs, our enemy is complexity.

Some tools for managing complexity:

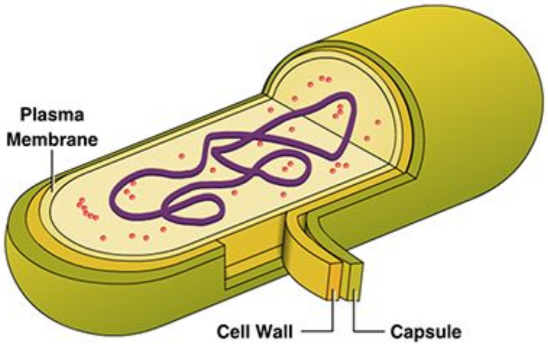
- Hierarchical abstraction.
  - Create **layers of abstraction**, with clear abstraction barriers!
- “Design for change” (D. Parnas)
  - Organize program around objects.
  - Let objects decide how things are done.
  - **Hide information** others don’t need.

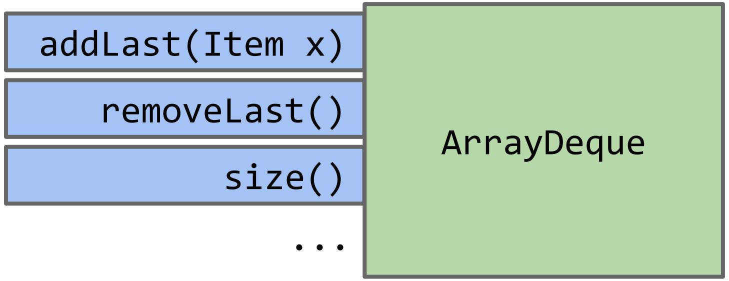
Managing complexity supremely important for large projects (e.g. project 2).

Modules and Encapsulation [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 <u>completely hidden</u>, and it can be accessed only through a documented interface.





A Cautionary Tale

Interesting forum questions from extra credit assignment from a few years ago.

How can we check the length of StudentArrayDeque?

I am trying to find a bug in the resizing method, but I don't know how to see the length of the StudentArrayDeque.

StudentArrayDeque.length() and StudentArrayDeque.length do not work…so I don't know how to check whether the Array can expand to double its capacity or not.

Private access in given classes

I wanted to test whether the resizing and downsizing is working properly, but when I try to call array.items.length, the compiler yells at me, saying items is a private variable. Is there any way around this, or should we just not test this?

Can we assume these things about StudentArrayDeque?

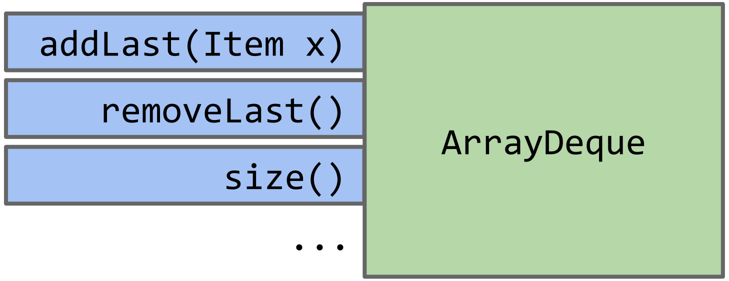
Can we assume the StudentArrayDeque implementation uses nextFront = 4, nextLast = 5, and starting size array 8?

Bottom line: Testing a Deque should usually not involve ANY assumptions about how it is implemented beyond what the public interface tells you.

Abstraction Barriers

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

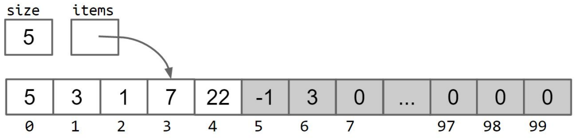
- Even when writing tests, you don't (usually) want to peer inside.



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

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

Implementation



size	items
5	

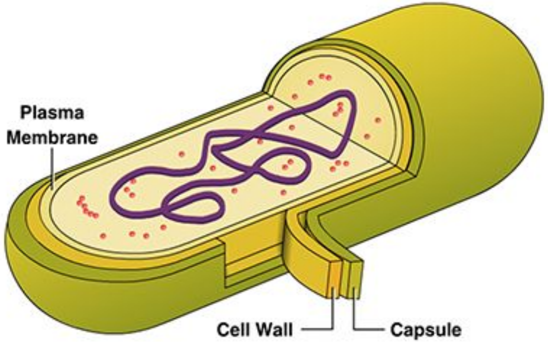
5	3	1	7	22	-1	3	0	...	0	0	0
0	1	2	3	4	5	6	7		97	98	99

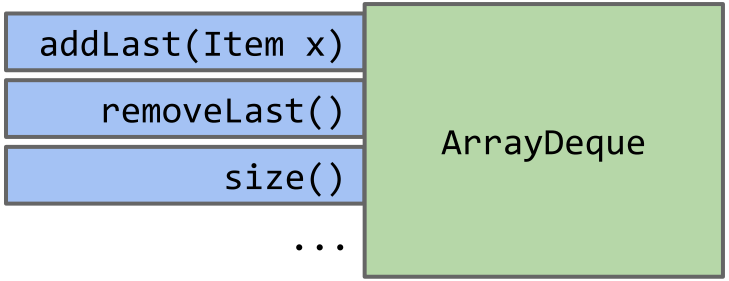
Modules and Encapsulation [Shewchuk]

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.
- As we'll see: Implementation inheritance (e.g. extends) breaks encapsulation!





Implementation Inheritance Breaks Encapsulation

Lecture 9, CS61B, Spring 2024

The Extends Keyword

- Rotating SLList
- Vengeful SLList
- A Boring Constructor Gotcha

Implementation Inheritance

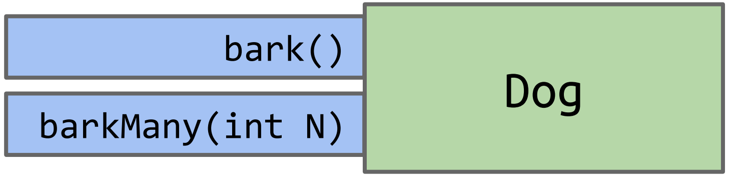
- The Object Class
- Is-A vs. Has-A, java.util.Stack
- Encapsulation
- **Implementation Inheritance Breaks Encapsulation**

Type Checking and Casting

Higher Order Functions in Java

Implementation Inheritance Breaks Encapsulation

Suppose we have a Dog class with the two methods shown.



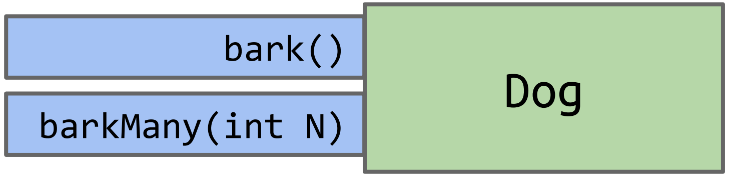
Dog.java

```
public void bark() {
   System.out.println("bark");
}

public void barkMany(int N) {
   for (int i = 0; i < N; i += 1) {
      bark();
   }
}
```

Implementation Inheritance Breaks Encapsulation

We could just as easily have implemented methods as shown below.

- From the outside, functionality is exactly the same, it’s just a question of aesthetics.



Dog.java

```java
public void bark() {
   barkMany(1);
}

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

http://yellkey.com/TODO

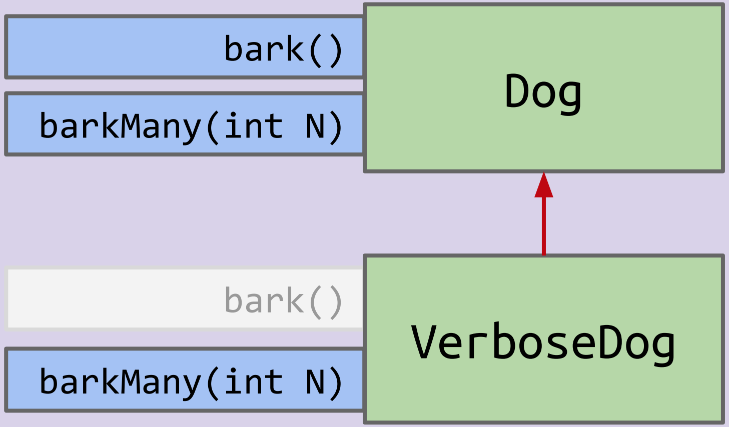
What would vd.barkMany(3) output?

a. As a dog, I say: bark bark bark
b. bark bark bark
c. Something else.

(assuming vd is a Verbose Dog)

Dog.java

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



VerboseDog.java

```
@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

Implementation Inheritance Breaks Encapsulation

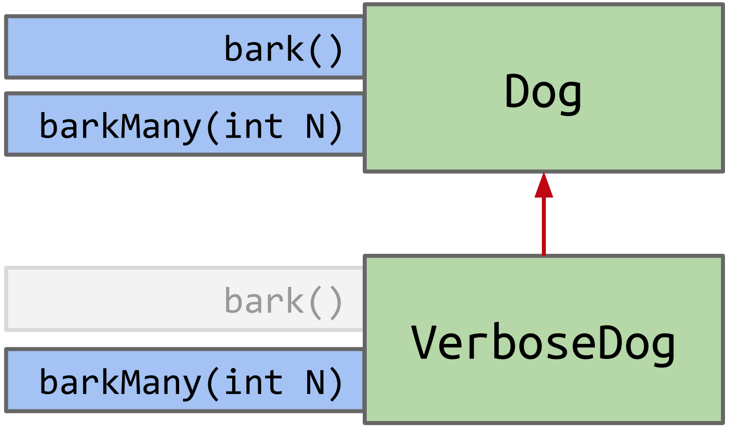
What would vd.barkMany(3) output?

- a. **As a dog, I say: bark bark bark**
- b. bark bark bark
- c. Something else.

(assuming vd is a Verbose Dog)

Dog.java

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



VerboseDog.java

```java
@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

http://yellkey.com/TODO

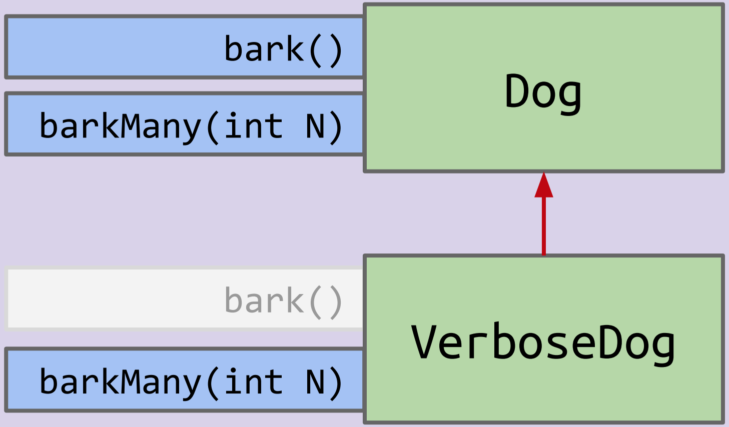
What would vd.barkMany(3) output?

a. As a dog, I say: bark bark bark
b. bark bark bark
c. Something else.

(assuming vd is a Verbose Dog)

Dog.java

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



VerboseDog.java

```
@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
   }
}
```

Implementation Inheritance Breaks Encapsulation

What would vd.barkMany(3) output?

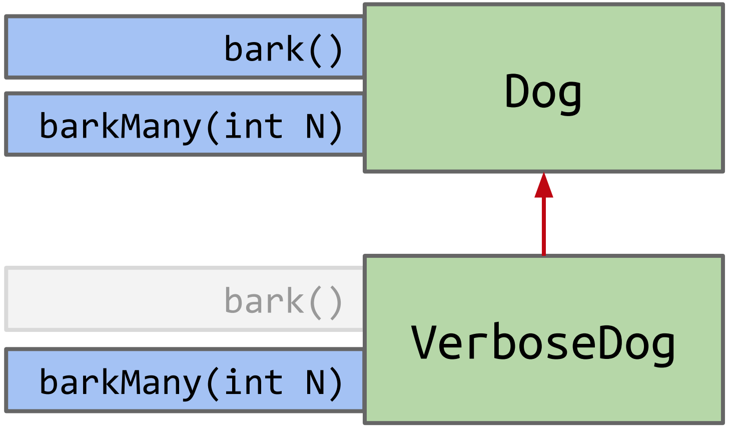
c. **Something else.**

- Gets caught in an infinite loop!

(assuming vd is a Verbose Dog)

Dog.java

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



VerboseDog.java

```java
@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

Lecture 9, CS61B, Spring 2024

The Extends Keyword

- Rotating SLList
- Vengeful SLList
- A Boring Constructor Gotcha

Implementation Inheritance

- The Object Class
- Is-A vs. Has-A, java.util.Stack
- Encapsulation
- Implementation Inheritance Breaks Encapsulation

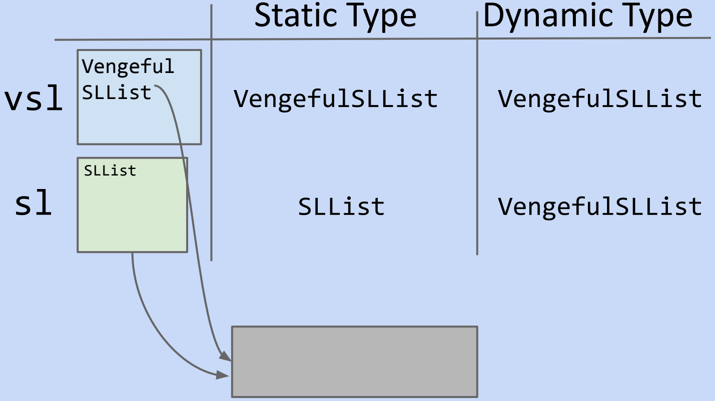
Type Checking and Casting

Higher Order Functions in Java

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;
}
```

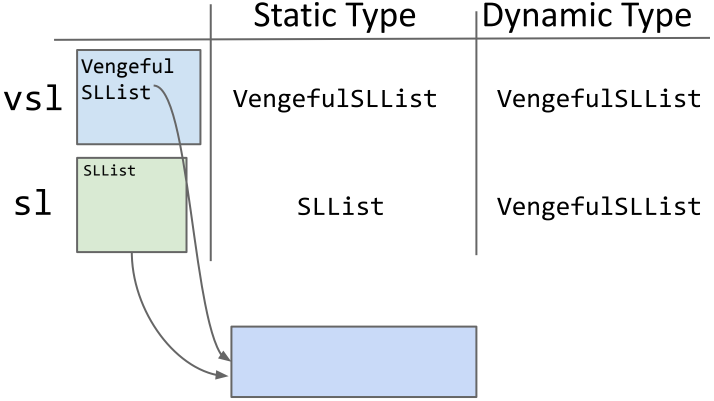
Reminder: VengefulSLList overrides removeLast and provides a new method called printLostItems.

Reminder: Dynamic Method Selection

Also called dynamic type.

If <u>overridden</u>, 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.

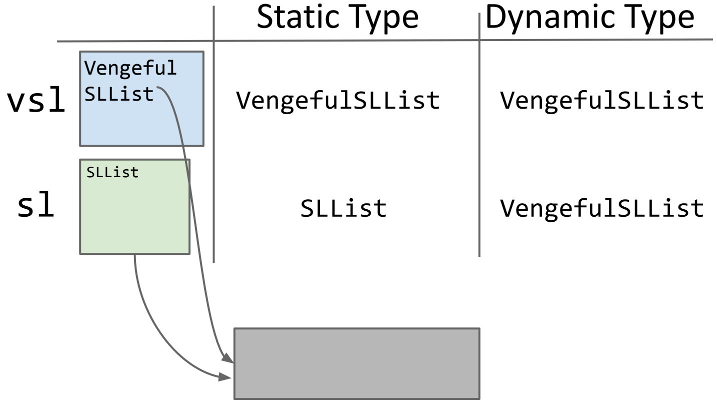
Reminder: VengefulSLList overrides removeLast and provides a new method called printLostItems.

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

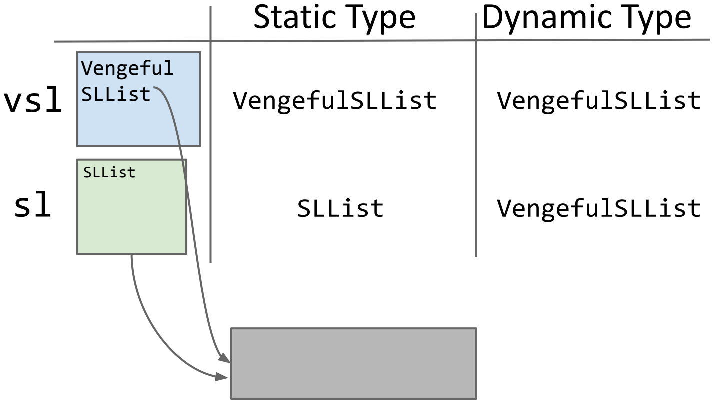
Also called static type.

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

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

```
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!



	Static Type	Dynamic Type
vsl	VengefulSLList	VengefulSLList
sl	SLList	VengefulSLList

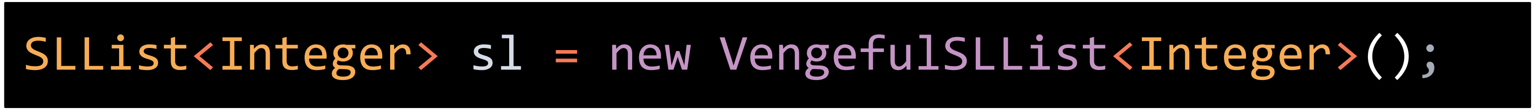
Compiler also allows assignments based on compile-time types.

- 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

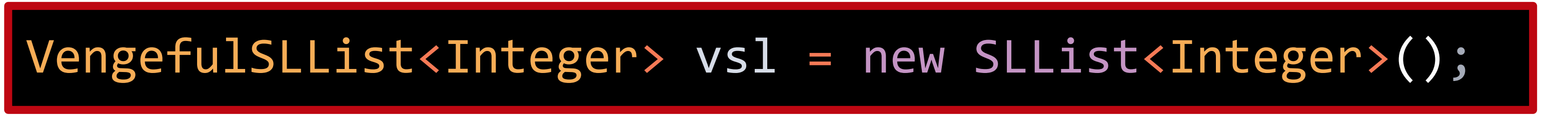
Expressions have compile-time types:

- An expression using the new keyword has the specified compile-time type.



```
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.



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

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

Compilation error!

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...

Higher Order Functions in Java

Lecture 9, CS61B, Spring 2024

The Extends Keyword

- Rotating SLList
- Vengeful SLList
- A Boring Constructor Gotcha

Implementation Inheritance

- The Object Class
- Is-A vs. Has-A, java.util.Stack
- Encapsulation
- Implementation Inheritance Breaks Encapsulation

Type Checking and Casting

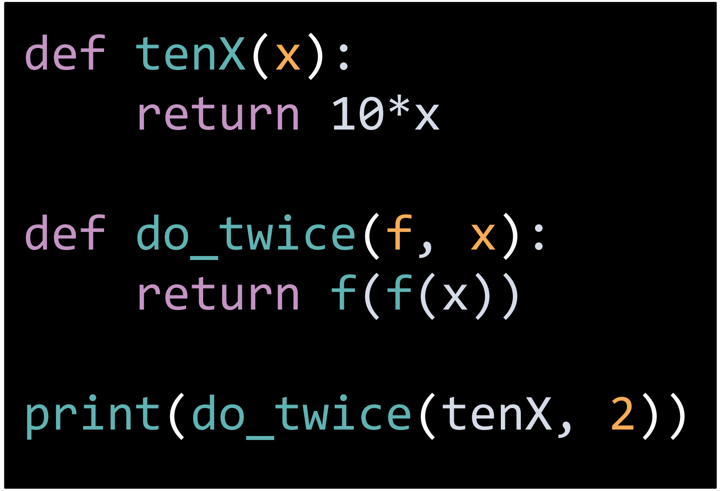
Higher Order Functions in Java

Higher Order Functions

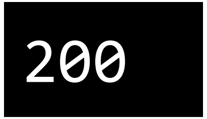
Higher Order Function: A function that treats another function as data.

- e.g. takes a function as input.

Example in Python:



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

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

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



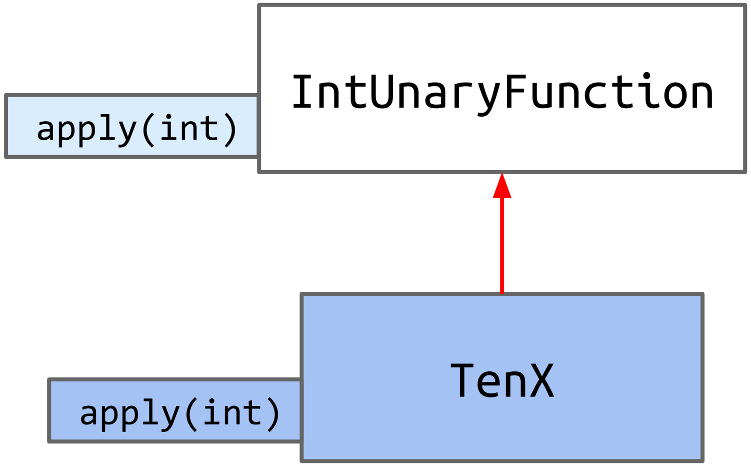
```
200
```

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.



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

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

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

Coding Demo: Higher-Order Function

IntUnaryFunction.java

```java
/** Represent a function that takes in an integer, and returns an integer. */
public interface IntUnaryFunction {

}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

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

Could say `public int apply` instead of `int apply`, but the `public` is redundant.

Coding Demo: Higher-Order Function

IntUnaryFunction.java

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

TenX.java

```java
public class TenX implements IntUnaryFunction {




}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

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

TenX.java

```java
public class TenX implements IntUnaryFunction {

   public int apply(int x) {

   }
}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

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

TenX.java

```java
public class TenX implements IntUnaryFunction {
   /** Returns ten times the argument. */
   public int apply(int x) {

   }
}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

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

TenX.java

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

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: Java code below is equivalent to given python code.

```java
public interface IntUnaryFunction {
   int apply(int x);
}
```

```java
public class TenX implements IntUnaryFunction {
   public int apply(int x) {
       return 10 * x;
   }
}
```

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

Coding Demo: Higher-Order Function

IntUnaryFunction.java

```java
public interface IntUnaryFunction {
   int apply(int x);
}
```

TenX.java

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

HoFDemo.java

```java
/** Demonstrates higher order functions in Java. */
public class HoFDemo {









}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

```java
public interface IntUnaryFunction {
  int apply(int x);
}
```

TenX.java

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

HoFDemo.java

```java
/** Demonstrates higher order functions in Java. */
public class HoFDemo {
  public static int doTwice(IntUnaryFunction f, int x) {

  }






}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

```java
public interface IntUnaryFunction {
    int apply(int x);
}
```

TenX.java

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

HoFDemo.java

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







}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

```java
public interface IntUnaryFunction {
   int apply(int x);
}
```

TenX.java

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

HoFDemo.java

```java
/** 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) {


   }
}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

```java
public interface IntUnaryFunction {
   int apply(int x);
}
```

TenX.java

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

HoFDemo.java

```java
/** 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) {

       System.out.println(doTwice(TenX, 2));
   }
}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

```java
public interface IntUnaryFunction {
   int apply(int x);
}
```

TenX.java

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

HoFDemo.java

```java
/** 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));
   }
}
```

Coding Demo: Higher-Order Function

IntUnaryFunction.java

```
public interface IntUnaryFunction {
   int apply(int x);
}
```

TenX.java

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

HoFDemo.java

```
/** 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)); // should print 200
   }
}
```

Example: Higher Order Functions Using Interfaces in Java

```java
public interface IntUnaryFunction {
   int apply(int x);
}
```

```java
public class TenX implements IntUnaryFunction {
   public int apply(int x) {
       return 10 * x;
   }
}
```

```java
public class HoFDemo {
   public static int do_twice(IntUnaryFunction f, int x) {
       return f.apply(f.apply(x));
   }
   public static void main(String[] args) {
       System.out.println(do_twice(new TenX(), 2));
   }
}
```

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

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

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

Example: Higher Order Functions in Java 8 or Later

In Java 8, new types were introduced: now can can hold references to methods.

- You’re welcome to use these features, but we won’t teach them.
- Why? The old way is still widely used, e.g. `Comparators` (see next lecture).

```java
public class Java8HofDemo {
   public static int tenX(int x) {
       return 10*x;
   }
   public static int doTwice(Function<Integer, Integer> f, int x) {
       return f.apply(f.apply(x));
   }
   public static void main(String[] args) {
       int result = doTwice(Java8HofDemo::tenX, 2);
       System.out.println(result);
   }
}
```

Implementation Inheritance Cheatsheet

VengefulSLList extends SLList means a VenglefulSLList 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 <u>overridden</u> methods the actual method invoked is based on **dynamic** type of invoking expression, e.g. Dog.maxDog(d1, d2).bark();
- Can use casting to overrule compiler type checking.

Does not apply to **overloaded** methods!