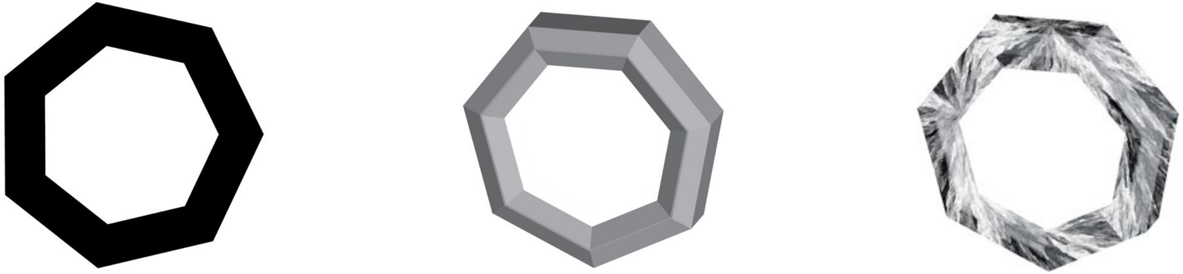


Lecture 8 (Inheritance 1)

# Interface and Implementation Inheritance

**CS61B, Spring 2024 @ UC Berkeley**

Slides credit: Josh Hug

# The Desire for Generality

Lecture 8, CS61B, Spring 2024

**Interface Inheritance**

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

Implementation Inheritance

- Default Methods
- Overriding Default Methods

Static and Dynamic Type

- Static and Dynamic Type
- Changes to Scope in 61B

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

```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, Spring 2024

**Interface Inheritance**

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

Implementation Inheritance

- Default Methods
- Overriding Default Methods

Static and Dynamic Type

- Static and Dynamic Type
- Changes to Scope in 61B

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

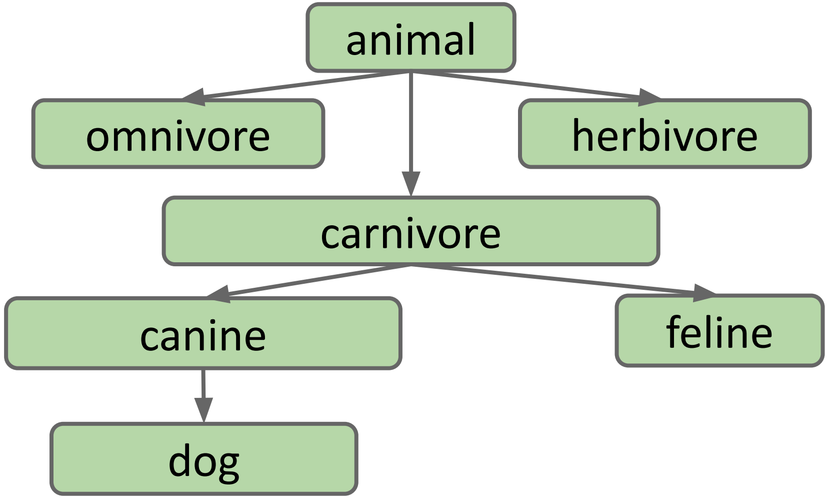
We use the word hyponym for the opposite type of relationship.

- “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, Spring 2024

## Interface Inheritance

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

Implementation Inheritance

- Default Methods
- Overriding Default Methods

Static and Dynamic Type

- Static and Dynamic Type
- Changes to Scope in 61B

Using Inheritance Safely

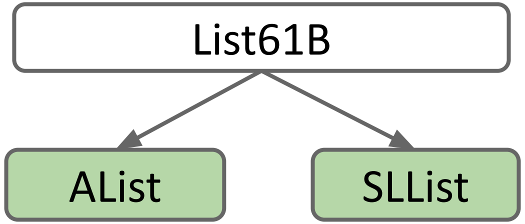
# Simple Hyponymic Relationships in Java

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 **what** a List is able to do, **not how** to do it.

List61B.java

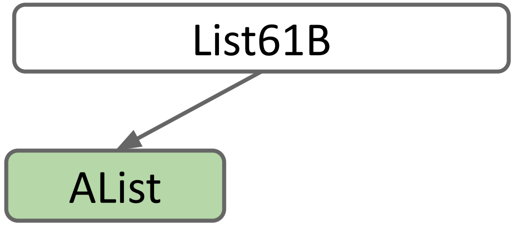
```
public interface List61B<Item> {
   public void insert(Item x, int position);
   public void addFirst(Item x);
   public void addLast(Item y);
   public Item getFirst();
   public Item getLast();
   public Item removeLast();
   public Item get(int i);
   public int  size();
}
```

# 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) {
      ...
```

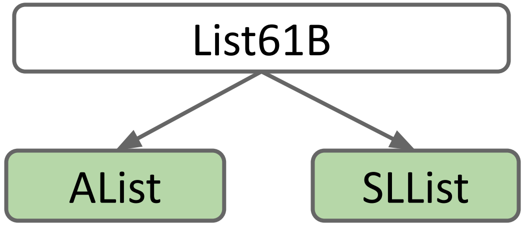


# Step 2: Implementing the List61B Interface

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:

```java
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);
}
```

```java
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

```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, Spring 2024

**Interface Inheritance**

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

Implementation Inheritance

- Default Methods
- Overriding Default Methods

Static and Dynamic Type

- Static and Dynamic Type
- Changes to Scope in 61B

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) {
      ...
```

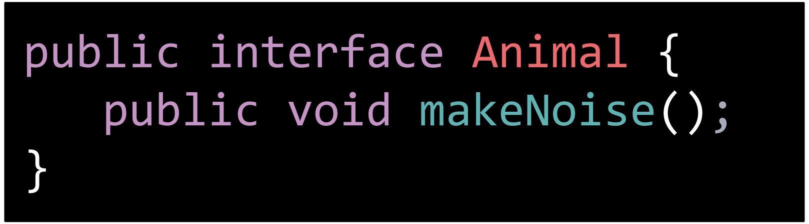
`AList` overrides `addLast(Item)`

# Method Overriding vs. Overloading

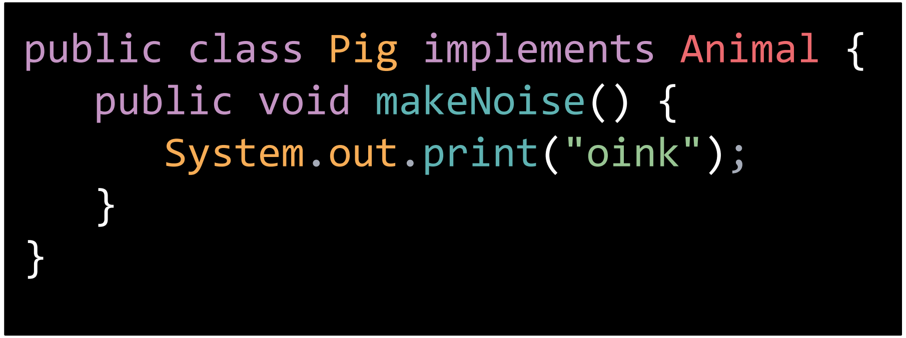
If a “subclass” has a method with the exact same signature as in the “superclass”, we say the subclass **overrides** the method.

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

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

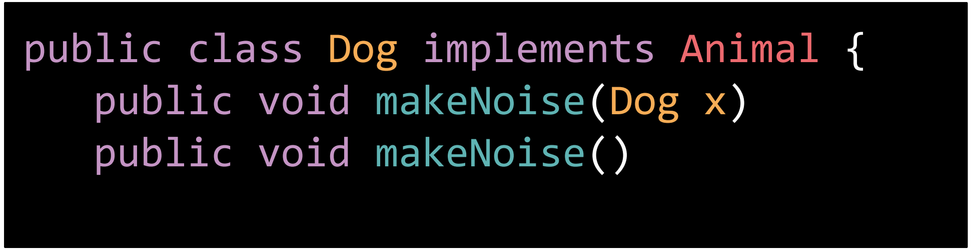


```
public interface Animal {
   public void makeNoise();
}
```



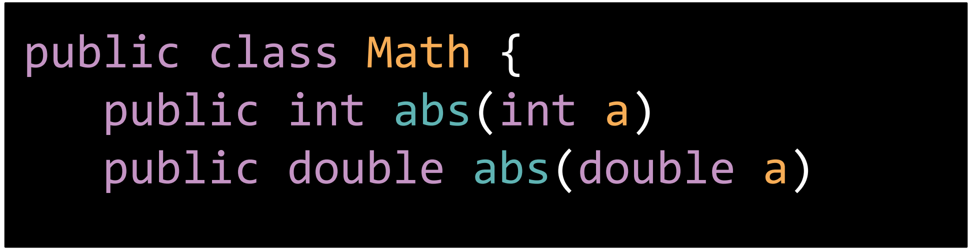
```
public class Pig implements Animal {
   public void makeNoise() {
      System.out.print("oink");
   }
}
```

`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)
```

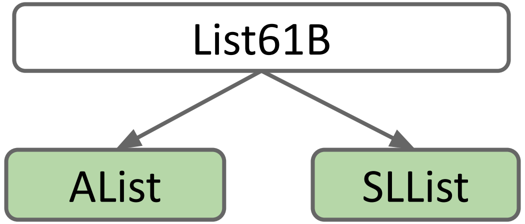
`abs` is overloaded

# 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, Spring 2024

**Interface Inheritance**

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

Implementation Inheritance

- Default Methods
- Overriding Default Methods

Static and Dynamic Type

- Static and Dynamic Type
- Changes to Scope in 61B

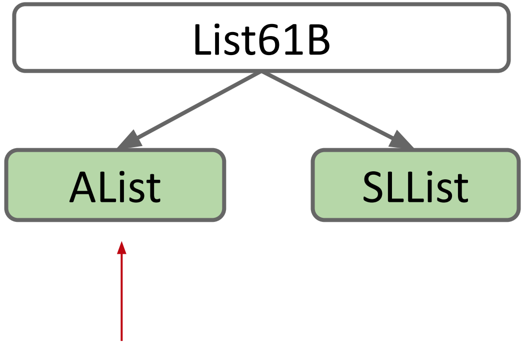
Using Inheritance Safely

# 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();
}
```

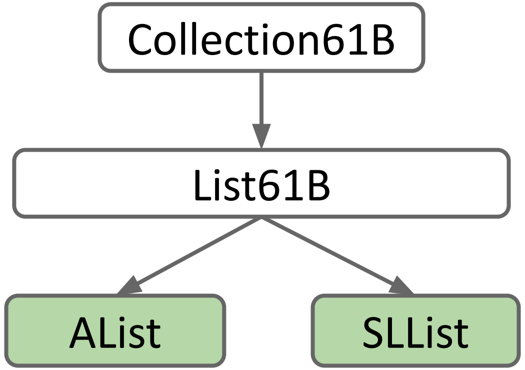


If `AList` doesn’t have a `proo()` method, `AList` will not compile!

# Interface Inheritance

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

- Interface: The list of all method signatures.
- Inheritance: The subclass "inherits" the interface.
- Specifies what the subclass can do, but not how.
- Subclasses must 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: yellkey.com/TODO

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");
}
```

# 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");
}
```

# Default Methods

Lecture 8, CS61B, Spring 2024

Interface Inheritance

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

**Implementation Inheritance**

- **Default Methods**
- Overriding Default Methods

Static and Dynamic Type

- Static and Dynamic Type
- Changes to Scope in 61B

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 **default** keyword to specify a method that subclasses should inherit from an **interface.**

- Example: Let’s add a default print() method to List61B.java

# Coding Demo: Default Methods

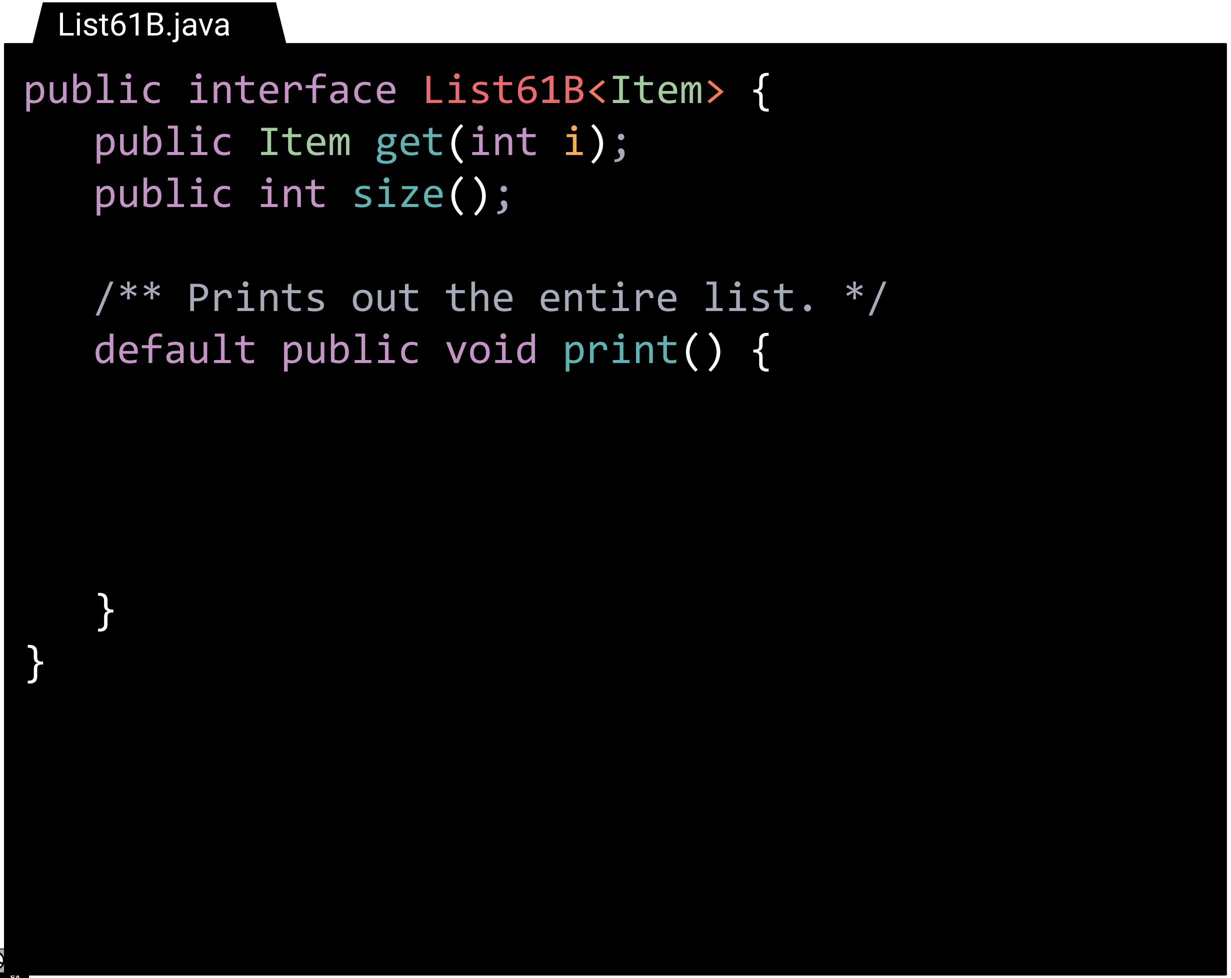
List61B.java

```java
public interface List61B<Item> {
   public Item get(int i);
   public int size();

   /** Prints out the entire list. */
   public void print() {




   }
}
```

If we try to write a method like we normally do in a class, we get an error:

"Interface methods cannot have body"

# Coding Demo: Default Methods

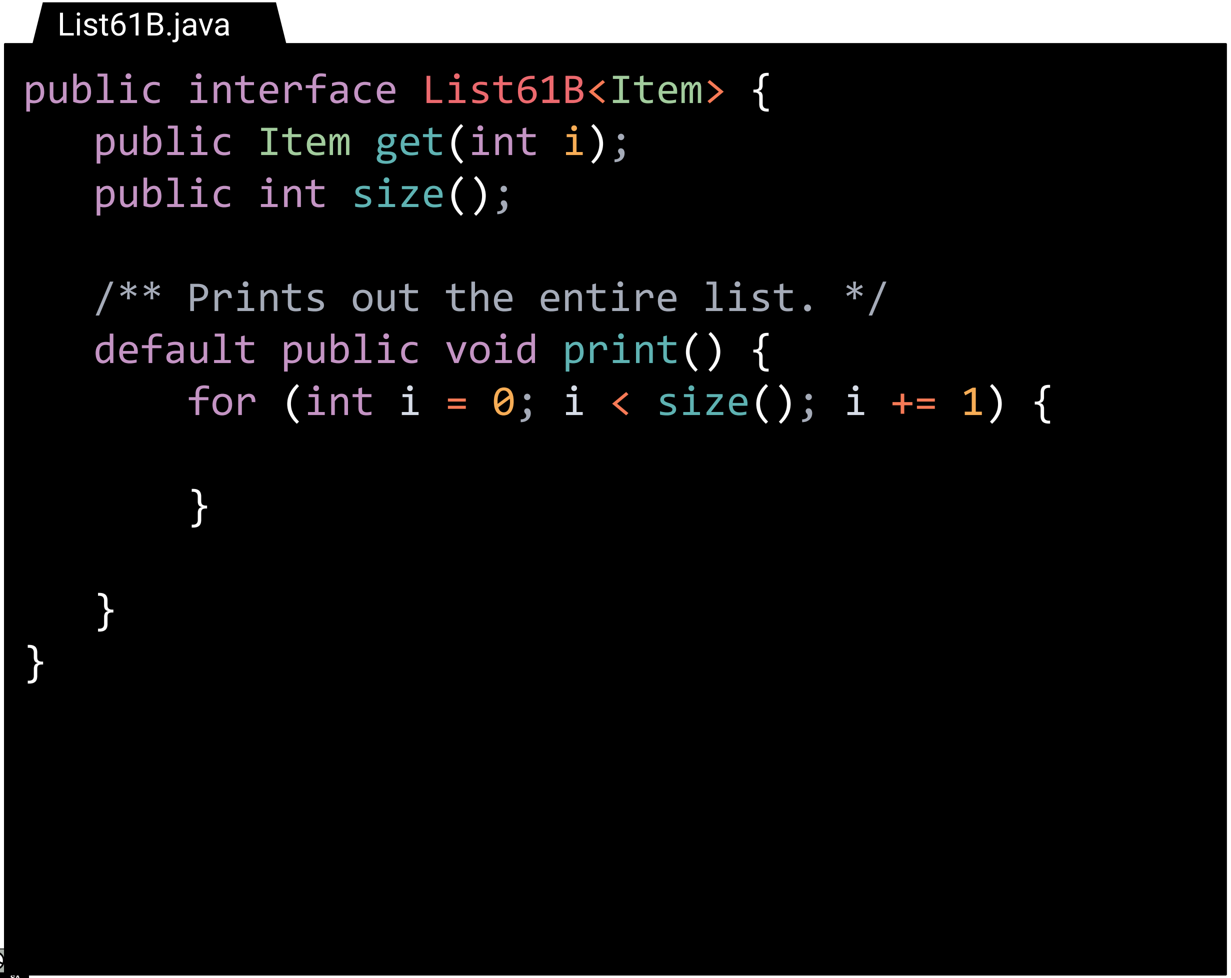


List61B.java

```java
public interface List61B<Item> {
   public Item get(int i);
   public int size();

   /** Prints out the entire list. */
   default public void print() {




   }
}
```

If we add the default keyword, the error goes away. Now we can write a method body in the interface.

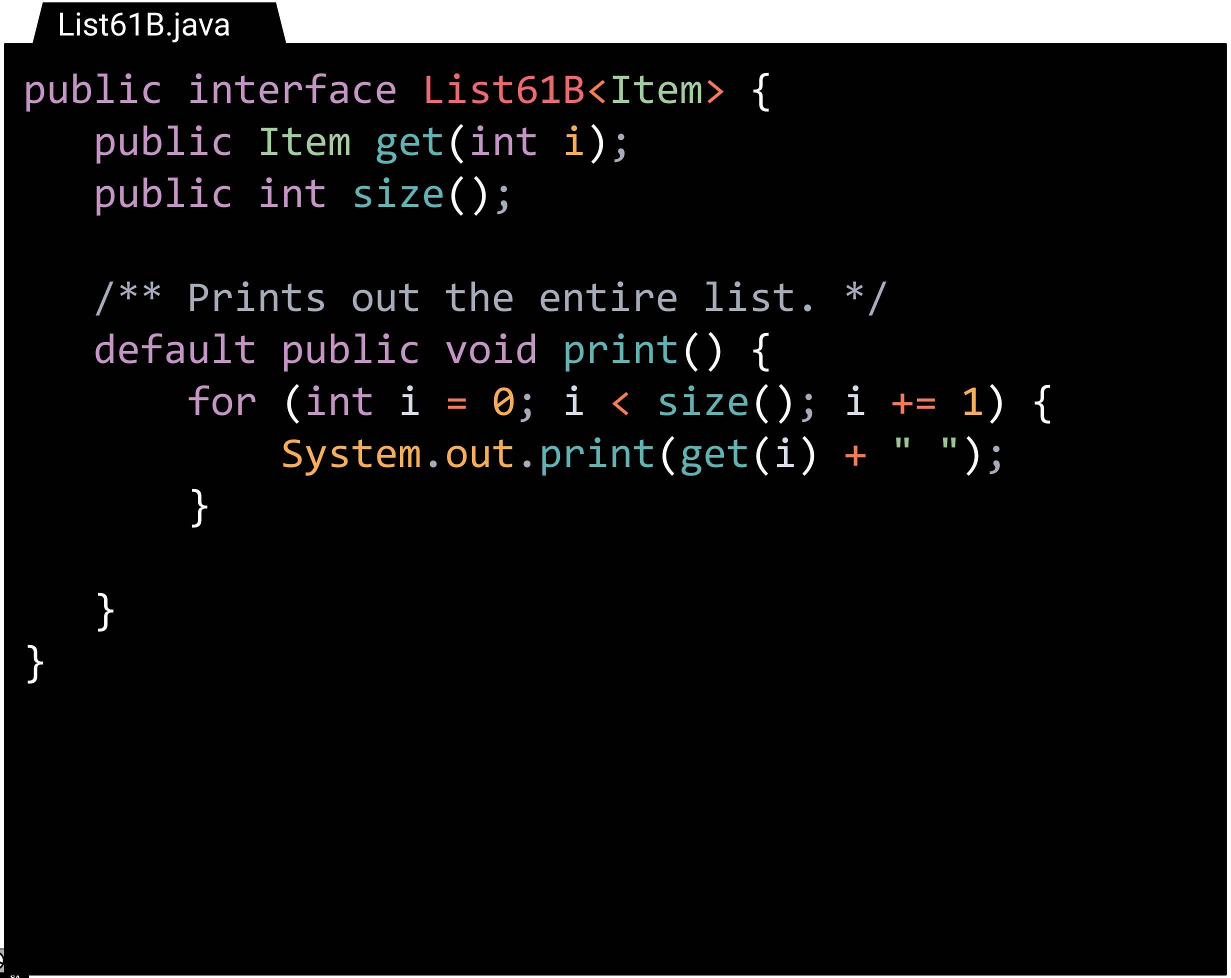
# Coding Demo: Default Methods



List61B.java

```java
public interface List61B<Item> {
    public Item get(int i);
    public int size();

    /** Prints out the entire list. */
    default public void print() {
        for (int i = 0; i < size(); i += 1) {

        }

    }
}
```

# Coding Demo: Default Methods



List61B.java

```java
public interface List61B<Item> {
   public Item get(int i);
   public int size();

   /** Prints out the entire list. */
   default public void print() {
       for (int i = 0; i < size(); i += 1) {
           System.out.print(get(i) + " ");
       }

   }
}
```

# Coding Demo: Default Methods

List61B.java

```java
public interface List61B<Item> {
    public Item get(int i);
    public int size();

    /** Prints out the entire list. */
    default public void print() {
        for (int i = 0; i < size(); i += 1) {
            System.out.print(get(i) + " ");
        }
        System.out.println();
    }
}
```

# Coding Demo: Default Methods

IsADemo.java

```
public class IsADemo {
   public static void main(String[] args) {
       List61B<String> someList = new SLList<>();
       someList.addFirst("elk");
       someList.addLast("dwell");
       someList.addLast("on");
       someList.addLast("existential");
       someList.addLast("crises");
       someList.print();
   }
}
```

```
elk dwell on
existential crises
```

SLLists don't have a print method, but the print method still works.

The default print method in the List61B interface is executed.

# Default Method Example: print()

```java
public interface List61B<Item> {
   public void insert(Item x, int position);
   public void addFirst(Item x);
   public void addLast(Item x);
   public Item getFirst();
   public Item getLast();
   public Item get(int i);
   public int size();
   public Item removeLast();
   default public void print() {
       for (int i = 0; i < size(); i += 1) {
           System.out.print(get(i) + " ");
       }
       System.out.println();
   }
}
```

# Question: yellkey.com/TODO

Is the print() method efficient?

a. Inefficient for AList and SLList
b. Efficient for AList, inefficient for SLList
c. Inefficient for AList, efficient for SLList
d. Efficient for both AList and SLList

```
public interface List61B<Item> {
    ...
   default public void print() {
       for (int i = 0; i < size(); i += 1) {
           System.out.print(get(i) + " ");
       }
      System.out.println();
   }
}
```

# Overriding Default Methods

Lecture 8, CS61B, Spring 2024

Interface Inheritance

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

**Implementation Inheritance**

- Default Methods
- **Overriding Default Methods**

Static and Dynamic Type

- Static and Dynamic Type
- Changes to Scope in 61B

Using Inheritance Safely

# Question

Is the print() method efficient?

a. Inefficient for AList and SLList

**b. Efficient for AList, inefficient for SLList**

c. Inefficient for AList, efficient for SLList

d. Efficient for both AList and SLList

```
public interface List61B<Item> {
    ...
    default public void print() {
        for (int i = 0; i < size(); i += 1) {
            System.out.print(get(i) + " ");
        }
        System.out.println();
    }
}
```

get has to seek all the way to the given item for SLLists.

# Coding Demo: Overriding Default Methods

SLList.java

```
public class SLList<Blorp> implements List61B<Blorp> {

   /** A print method that overrides
    *  List61B's inefficient print method. */

   public void print() {




   }
}
```

# Coding Demo: Overriding Default Methods

SLList.java

```java
public class SLList<Blorp> implements List61B<Blorp> {

   /** A print method that overrides
    *  List61B's inefficient print method. */
   @Override
   public void print() {




   }
}
```

# Coding Demo: Overriding Default Methods

SLList.java

```java
public class SLList<Blorp> implements List61B<Blorp> {

   /** A print method that overrides
    *  List61B's inefficient print method. */
   @Override
   public void print() {

      for (Node p = sentinel.next; p != null; p = p.next) {

      }
   }
}
```

# Coding Demo: Overriding Default Methods

SLList.java

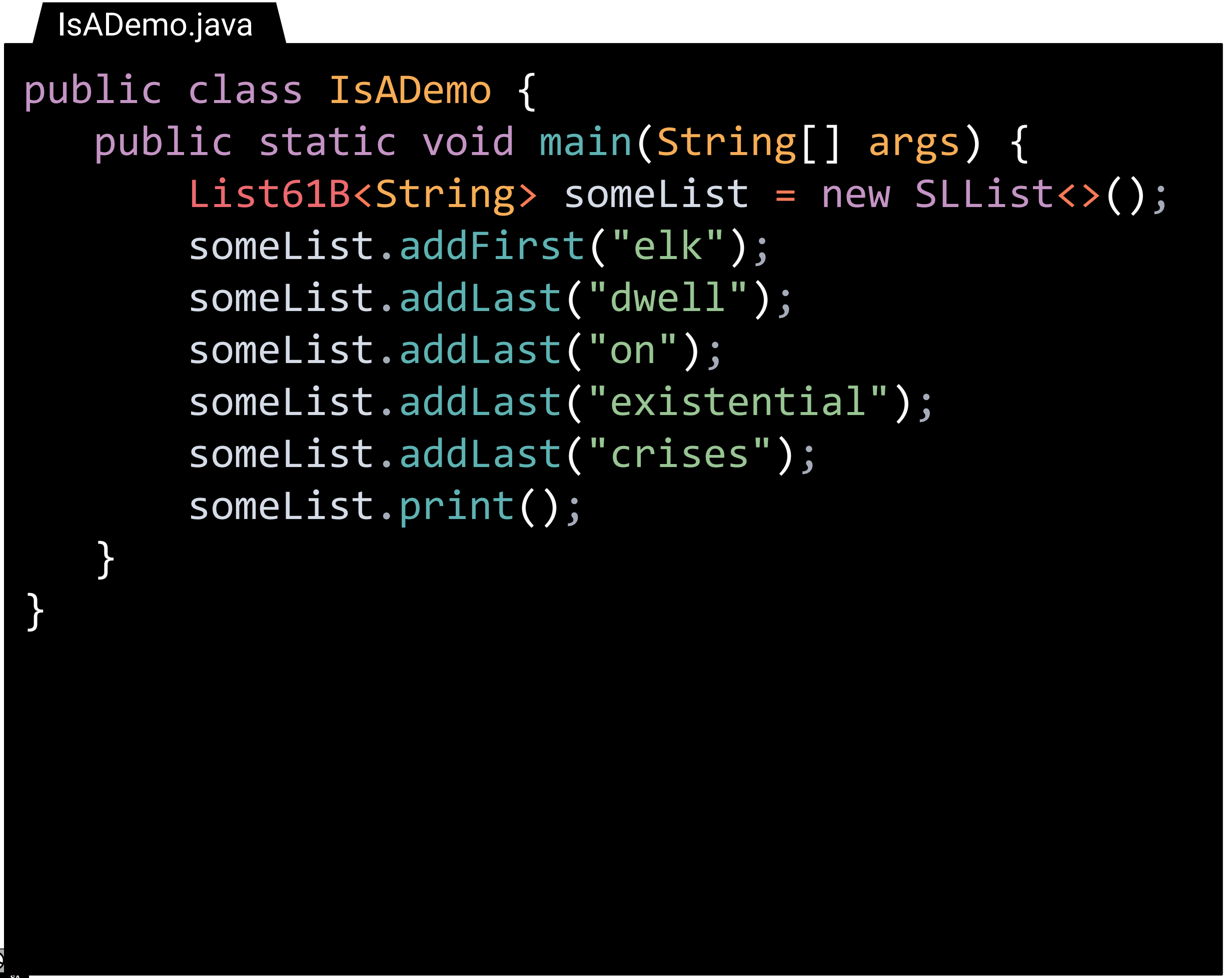
```java
public class SLList<Blorp> implements List61B<Blorp> {

   /** A print method that overrides
    *  List61B's inefficient print method. */
   @Override
   public void print() {

      for (Node p = sentinel.next; p != null; p = p.next) {
         System.out.print(p.item + " ");
      }
   }
}
```

# Coding Demo: Overriding Default Methods

SLList.java

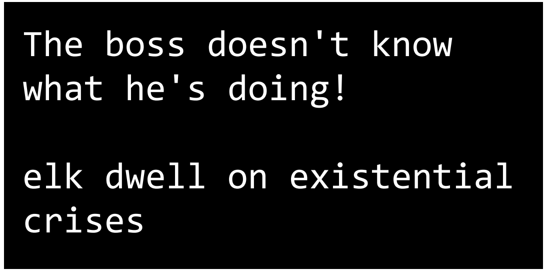
```java
public class SLList<Blorp> implements List61B<Blorp> {

   /** A print method that overrides
    *  List61B's inefficient print method. */
   @Override
   public void print() {
      System.out.println("The boss doesn't know what he's doing!");
      for (Node p = sentinel.next; p != null; p = p.next) {
         System.out.print(p.item + " ");
      }
   }
}
```

# Coding Demo: Default Methods



IsADemo.java

```java
public class IsADemo {
    public static void main(String[] args) {
        List61B<String> someList = new SLList<>();
        someList.addFirst("elk");
        someList.addLast("dwell");
        someList.addLast("on");
        someList.addLast("existential");
        someList.addLast("crises");
        someList.print();
    }
}
```



```
The boss doesn't know
what he's doing!

elk dwell on existential
crises
```

Now we're running the print method in SLList, not the print method in List61B.

# Overriding Default Methods

If you don’t like a default method, you can override it.

- Any call to print() on an SLList will use this method instead of default.
- Use (optional) @Override to catch typos like **`public void pirnt()`**

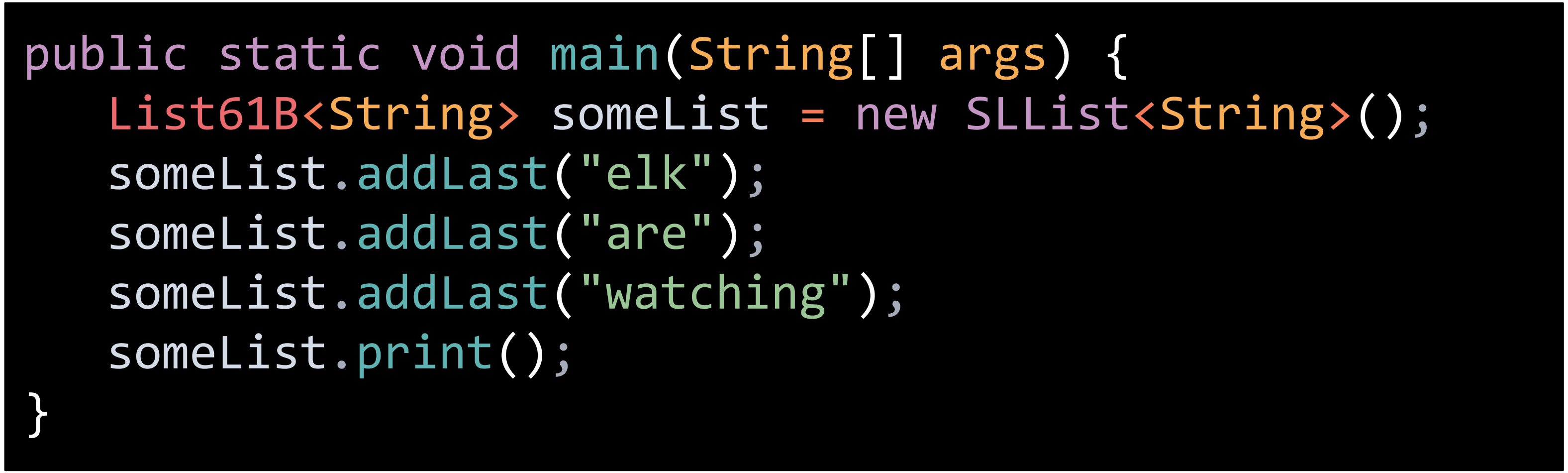
```
public class SLList<Blorp> implements List61B<Blorp> {
   @Override
   public void print() {
      for (Node p = sentinel.next; p != null; p = p.next) {
         System.out.print(p.item + " ");
      }
      System.out.println();
   }
}
```

## 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?

- List.print()
- SLList.print()



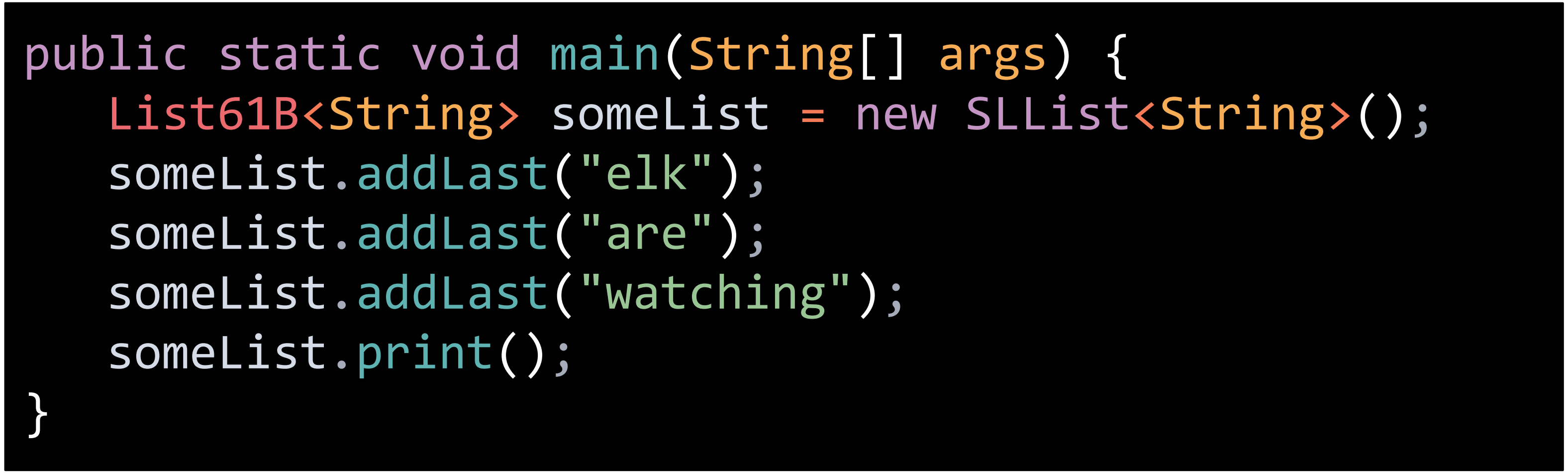
```java
public static void main(String[] args) {
   List61B<String> someList = new SLList<String>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   someList.print();
}
```

# 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?

- List.print()
- **SLList.print() : 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) {
   List61B<String> someList = new SLList<String>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   someList.print();
}
```

# Static and Dynamic Type

Lecture 8, CS61B, Spring 2024

Interface Inheritance

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

Implementation Inheritance

- Default Methods
- Overriding Default Methods

**Static and Dynamic Type**

- **Static and Dynamic Type**
- Changes to Scope in 61B

Using Inheritance Safely

# Static Type vs. Dynamic Type

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

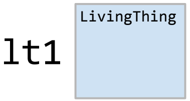
- This is the type specified at **declaration**. Never changes!

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”.

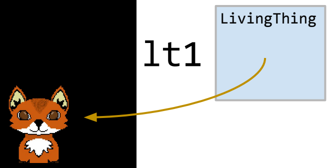
- This is the type specified at **declaration**. Never changes!

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”.

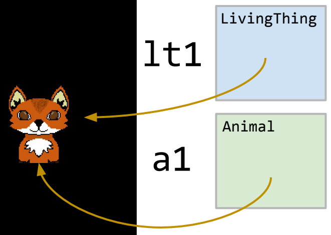
- This is the type specified at **declaration**. Never changes!

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

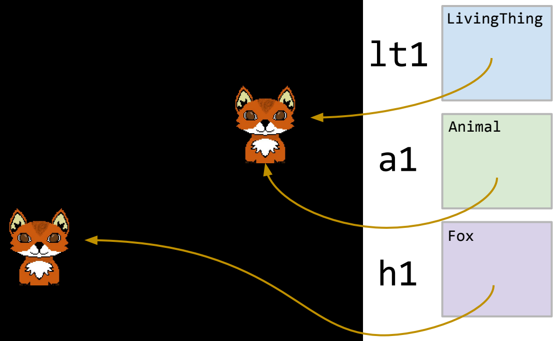
Every variable in Java has a “compile-time type”, a.k.a. “static type”.

- 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 |

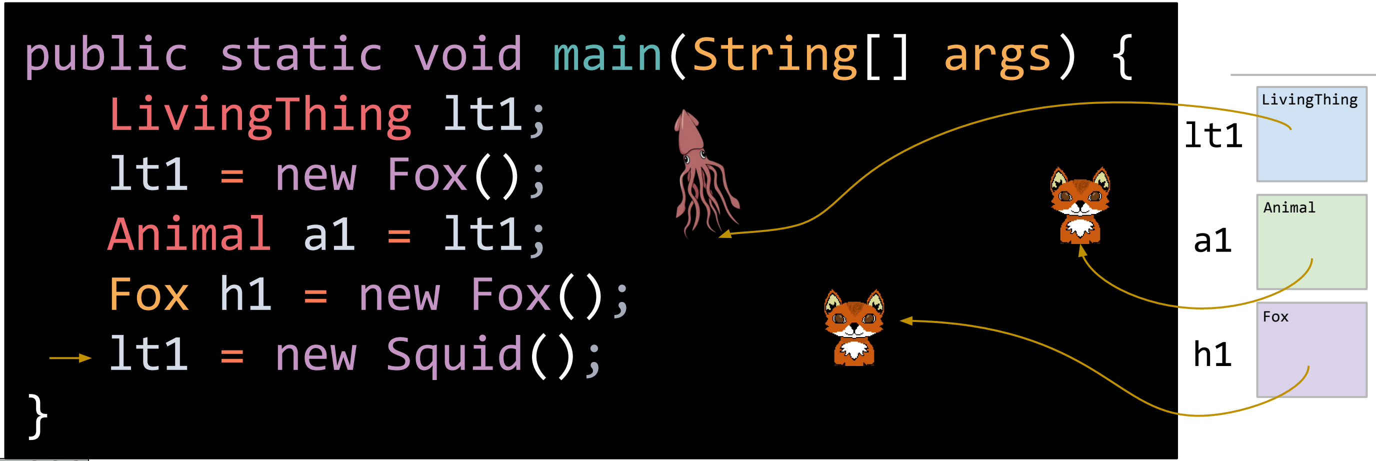
# Static Type vs. Dynamic Type

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:

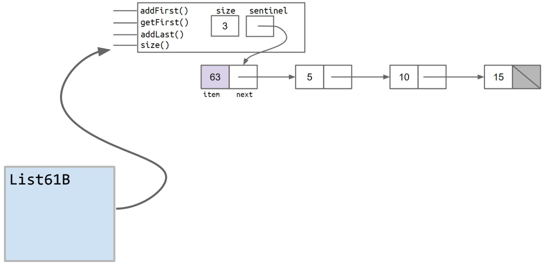
- compile-time type X
- run-time type Y

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 `List61B`

Dynamic Type `SLList`

# Changes to Scope in 61B

Lecture 8, CS61B, Spring 2024

Interface Inheritance

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

Implementation Inheritance

- Default Methods
- Overriding Default Methods

**Static and Dynamic Type**

- Static and Dynamic Type
- **Changes to Scope in 61B**

Using Inheritance Safely

# Older Versions of 61B (pre-2018)

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.

# Using Inheritance Safely

Lecture 8, CS61B, Spring 2024

Interface Inheritance

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

Implementation Inheritance

- Default Methods
- Overriding Default Methods

Static and Dynamic Type

- Static and Dynamic Type
- Changes to Scope in 61B

**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: print() implemented in List61B.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 two interfaces both give conflicting default methods?
- Encourages overly complex code (especially with novices).
  - Common mistake: Has-a vs. Is-a!
- Breaks encapsulation!
  - What is encapsulation? See next week.