Lecture 12

Midterm Review

CS 61B, Spring 2025 @ UC Berkeley

Review Lecture Topics Requested

- 61A-style coding questions (Java Syntax)
- Inheritance (Overloading/Overriding/Implements)
- Different forms of Lists (DLLists in particular)
- Comparators and Comparables (x2)
- Generics
- Inner Classes
- Class Inheritance, Interfaces
- Comparators vs Comparables
- Golden Rule of Equals
- Linked Lists
- Projects
- Sentinels
- Generics
- Static Classes/Methods
- Compiler vs Runtime Errors
- DLists

Brief Review

Lecture 12, CS61B, Spring 2025

Topic Review

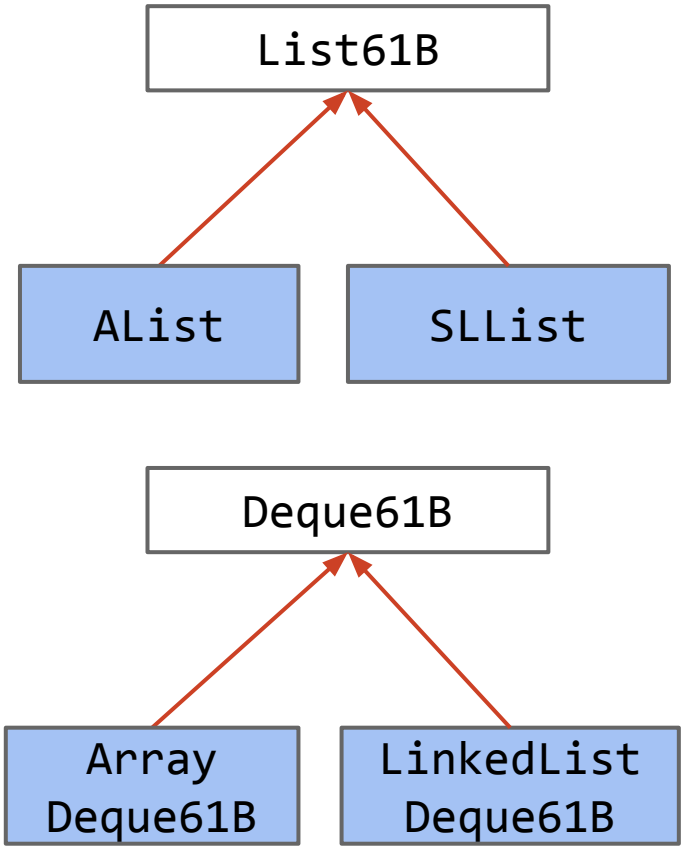
Let’s do a brief review of the topics you requested from the class.

- Feel free to interrupt and ask questions.

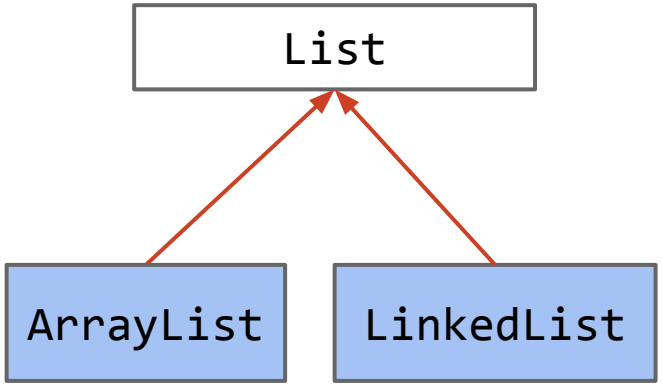
Abstract Data Types That We Built

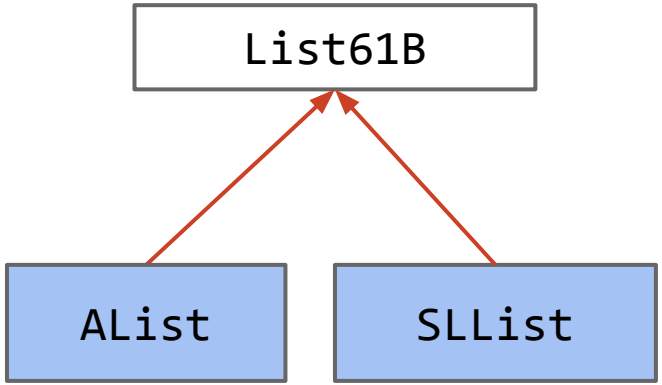
Abstract data types are defined by their operations.

- Examples: List61B, Deque61B



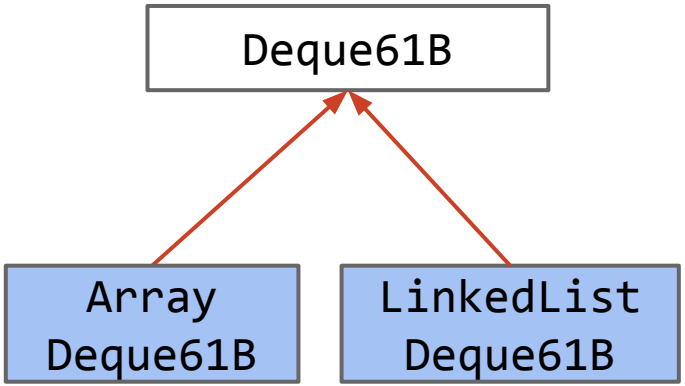
Abstract Data Types in Java





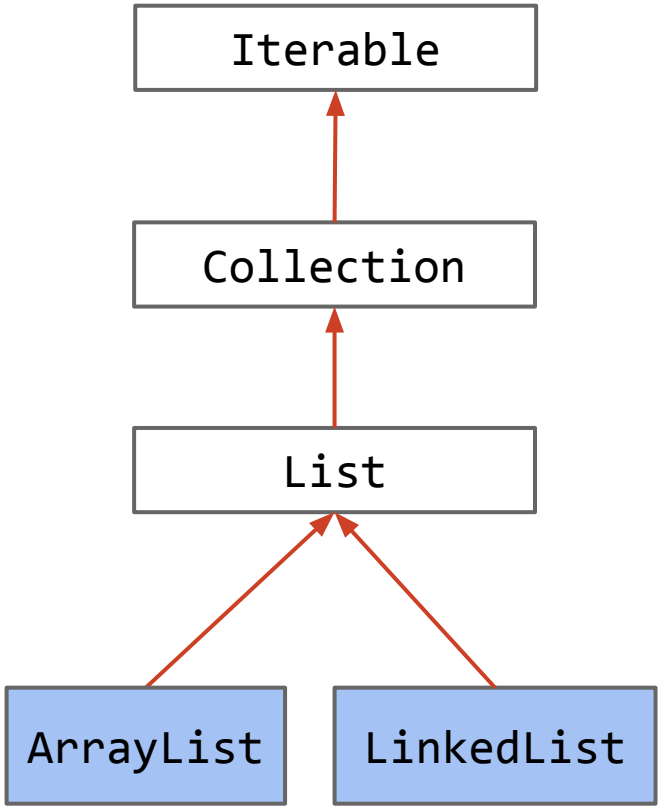
Java has its own built-in list type.

- Underlying implementations are pretty similar to what you did for for the Deque61B interface.

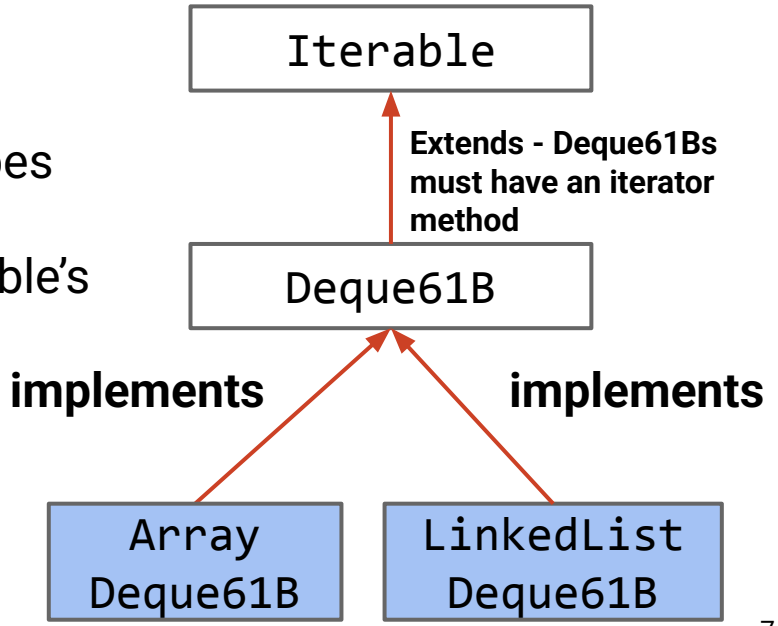


Interface Hierarchies

Interfaces can be part of a hierarchy, e.g. `Deque61B<T> extends Iterable<T>`



Deque61B does not need to override Iterable's methods†



Default Methods

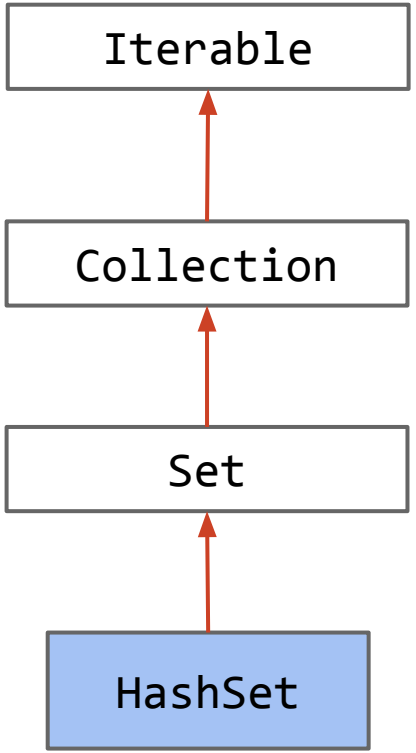
If there are methods that have natural default implementations in terms of other methods, we can write a default method. Example from List.java (https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/List.java):

```
default void addFirst(E e) {
    this.add(0, e);
}
```

Iterables

An iterable can be iterated over using a for-each loop.

```
Set<Integer> javaset = new HashSet<>();
javaset.add(5);
javaset.add(23);
javaset.add(42);
for (int i : javaset) {
   System.out.println(i);
}
```



The Iterable Interface

An Iterable can be iterated over using a for-each loop.

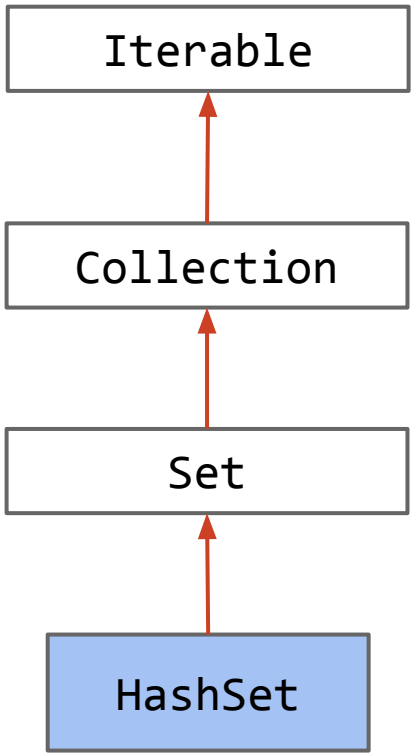
- An Iterable needs to be able to provide an Iterator.

The Iterable<T> interface:

- `public Iterator<T> iterator()`

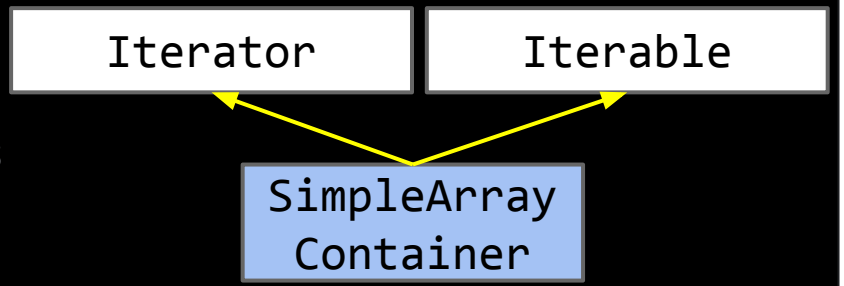
The Iterator<T> interface:

- `public boolean hasNext()`
- `public T next()`



Classes Can Implement Multiple Interfaces

```
public class SimpleArrayContainer<T> implements Iterable<T>, Iterator<T> {
    private final T[] items;
    private int currentIndex = 0;
    public SimpleArrayContainer(T[] items) {
        this.items = items;
    }
    public Iterator<T> iterator() {
        // Reset the position each time iteration starts
        currentIndex = 0;
        return this;
    }
    public boolean hasNext() {
        return currentIndex < items.length;
    }
    public T next() {
        T itemToReturn = items[currentIndex];
        currentIndex += 1;
        return itemToReturn;
    }
}
```



Comparables and Comparators

The `Comparable<T>` interface is used to compare `this` to another thing:

- `public int compareTo(T anotherThing)`

The `Comparator<T>` interface:

- `public int compare(T thing1, T thing2)`

Question: Since we're comparing two other objects (and not to `this`), why isn't the compare method static?

- You can't inherit and override static methods.
- Maybe we want a parameterized Comparator, e.g. CompareBasedOnNumberOfLetterOccurences(char c), and it compares based on how many c characters there are.

How do we specify generics?

Specify: If we mean pick a specific type, we ONLY do this for classes which have a <Blerf> at the top, example:

Declaration:

```
public class OurList61B<Blerf> {
}
```

Specification (a.k.a. usage):

```
OurList61B<String> alist = new OurList61B<>();
```

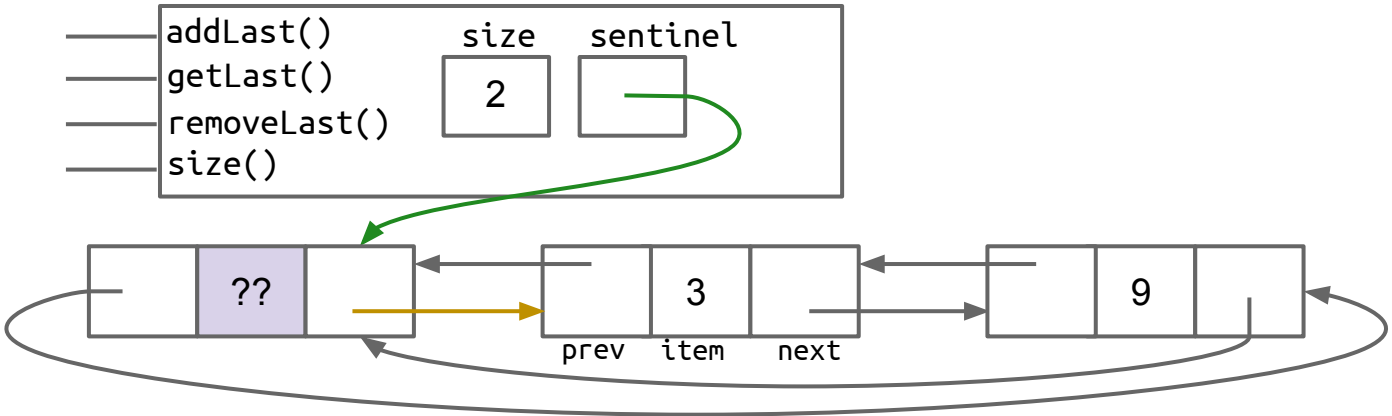
Linked List Implementations

Linked list based data structures are recursive data structures.

- You get more memory boxes to store stuff by creating new node objects.
- Each node object contains data and links to other nodes.
- Backwards links allow fast operations on both ends.

So far, we've seen singly linked and doubly linked lists.

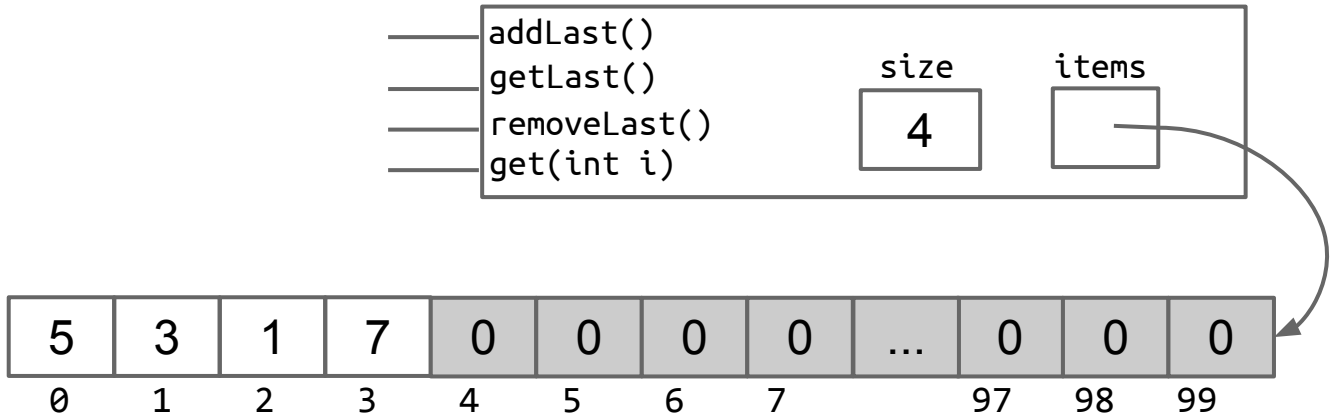
- Later we'll see trees and graphs.



Array Based Implementations

Array data structures are non-recursive data structures.

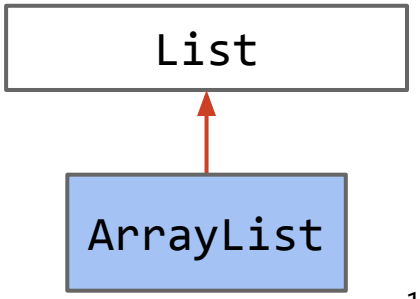
- You get more memory boxes by creating longer arrays.
- Resizing should be done geometrically, not arithmetically.
- Circularity allows fast operations on both ends. There are other non-circularity based approaches which are fast.



Syntax: For Inheritance

In Java, an interface enumerates all methods that a subclass must override.

- If a subclass fails to override a method, it will not compile.
  - Example: "`Implements Comparable`" but you don't override `compareTo`
- It's OK to have additional methods.
  - Example: "`Implements Comparable`" and you also have a `compare` method.
- It's OK to overload methods with the same name as the interface method.
  - `get(int i, int j)` is fine, even if interface has `get(int i)`
- Recommended, but not required, to use @Override annotation.



Terminology

A **member** of a class is:

- A variable.
- A method.
- A nested class.

Subclasses inherit all members of their superclasses.

- So far we've only talked about superclasses which are interfaces.

Syntax: Static vs. Non-Static

A **static** member of class X is associated with class X, rather than an instance of class X. We can think of this as "there is no instance of X called `this`".

```
public class Dog {
    public int size;
    public static void bark() {
        if (size > 0) {
            System.out.println("bark");
        }
    }
}
```

Since we don't specify where size comes from, Java implicitly checks to see if `this.size` exists. It does not, **there is no `this`**.

Syntax: Static vs. Non-Static

A **static** member of class X is associated with class X, rather than an instance of class X. We can think of this as “there is no instance of X called `this`”.

Question: If this variable was static, would this work? Java says “i have no local variable called size”, but is there a “this.size?”, no ok is there a “Dog.size”? Yes

IT WORKS, but you should write as Dog.size

```
public class Dog {
    public static int size;
    public static void bark() {
        if (size > 0) {
            System.out.println("bark");
        }
    }
}
```

Syntax: Static vs. Non-Static

A **static** member of class X is associated with class X, rather than an instance of class X. We can think of this as “there is no instance of X called `this`”.

Question: If this variable was static, would this work? Java says “i have no local variable called size”, but is there a “this.size?”, no ok is there a “Dog.size”? yes

IT WORKS, but you should write as Dog.size

```
public class Dog {
    public static int size;
    public void bark() {
        if (size > 0) {
            System.out.println("bark");
        }
    }
}
```

Syntax: Static vs. Non-Static

A **static** member of class X is associated with class X, rather than an instance of class X. We can think of this as “there is no instance of X called `this`”.

- Non-static members **cannot** be accessed from static members.
- Static members can be accessed by non-static members.
- **If you don’t use this. or Dog., java will try those, and if either works, great!**

Syntax: Static vs. Non-Static

A **static** member of class X is associated with class X, rather than an instance of class X. We can think of this as "there is no instance of X called `this`".

- Non-static members **cannot** be accessed from static members.
- Static members can be accessed by non-static members.
- If you don't use this. or Dog., java will try those, and if either works, great!

Question:

- What's an example of a nested class we've seen that is non-static? Why was it non-static?
  - ArrayDequeIterator, needs to be able toa ccess the enclosing class's variables. You have to ber able to access ArrayDeque's size and items to iterate over them.
- What's an example of a nested class we've seen that is static? Why was it static?
  - Node in AList., because it doesn't need to know about the bigger picture.

If you wanted ot make ArrayDequeIterator static, could you pass in the specific instance variables it needs to know?

- That will work.

If we are working in a nested class, say ArrayDequeIterator, and we use “this”, how do we know if this refers to the iterator object or the ArrayDeque enclosing it.

- If it’s a static iterator class, this must be referring to the iterator.
- If it’s a non-static iterator class, I forgot.

THE MOST IMPORTANT THING ABOUT ALL THIS

A static nested class cannot access the instance variables of an instance of the outer class.

- **An ALIst node (from lecture) cannot access the AList's size.**

Aret here weird edge cases and crazy things we might do?

- Yes, but I won't test you on them/

Compilation vs. Interpretation

In Java, compilation and interpretation are two separate steps.

Why make a class file at all?

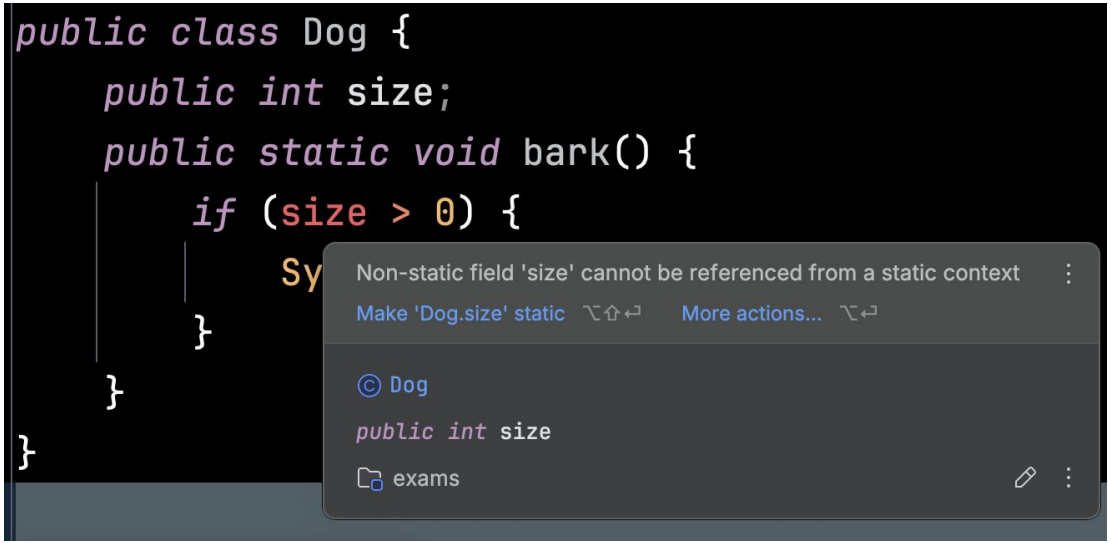
- .class file has been type checked. Distributed code is safer.
- .class files are ‘simpler’ for machine to execute. Distributed code is faster.

You don’t see this process in 61B because IntellIJ compiles in the background secretly.

Java: Compilation vs. Runtime

Java's type checking is done before you run the code. Will not compile otherwise.

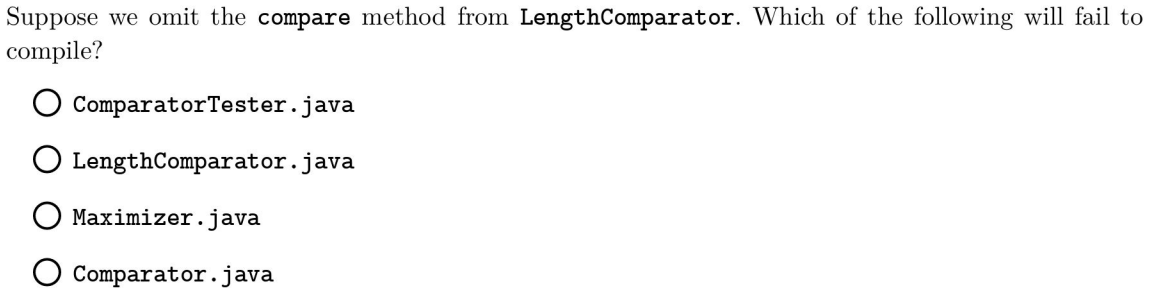
- Example below, this code is not compiling! No .class file is being generated.



Java: Compilation vs. Runtime Errors

Java's type checking is done before you run the code. Will not compile otherwise.

- Example: When you pass something to a method, Java makes sure the thing you are passing makes sense.
  - Best practice for this is the discussion worksheet problem that has questions like this:


Suppose we omit the `compare` method from `LengthComparator`. Which of the following will fail to compile?

- ◯ `ComparatorTester.java`
- ◯ `LengthComparator.java`
- ◯ `Maximizer.java`
- ◯ `Comparator.java`

Runtime errors that are (usually) impossible in Java:

- Method is missing (e.g. you forgot to declare a compareTo method).
- Undefined variables (Java won't let you access them).
- Wrong type of object passed (you passed a Dog, we expected a List).

Things Not In Scope

These things are not in scope, but might appear on old exams:

- Classes extending classes.
- Mixes of overloaded and overridden methods with the same name but different signatures.
- Dynamic method selection.
- Static (a.k.a. compile time) vs. dynamic (a.k.a. run time) type.

Default methods in interfaces - can you use this keyword?

- Probably?

Implements vs. extends

- We use impleemnts to say that a CLASS **implements** an INTERFACE. (ArrayDeque implements Deque)

- We use extends to say that an INTERFACE has all the properties of some super-interface (example: Deque extends Iterable, this means all Deques are iterables)

Is there a difference between

Whyat’s up with

```
public static <T> T max(Iterable<T> iterable, Comparator<T> comp)
```

We sometimes want static methods that can take any type.

- The problem is that such static methods won’t have an instance that provides the specific type. That is, we don’t have like a ArrayList<String> as = new Arraylist<>() anywhere.
- So the fix in Java is to make the method itself generic by putting that annoying <T> thing after static.
  - Thist ells JAa that this method can take ANY type.

Solving Some Problems

Lecture 12, CS61B, Spring 2025

Exam Problems

I'll solve some problems cold from here. Hopefully seeing my thinking process is useful.