Collections
Part 1: Lists

Today's Objectives

- Collections - why and how do we use them?
- List vs array
- Understand the concept of code libraries and namespaces
- Use the for-each loop to iterate through a collection
- Stacks* and Queues*
- Java Wrapper Objects

*Not included in lecture but good to know

Arrays Recap

- Arrays let us work with a collection of like data types.
- Arrays aren’t very flexible.
- Collections provide much more flexible ways to work with collections of data.

Introducing: Collections

- A collection represents a group of objects, known as its elements.
  - Some collections allow duplicate elements and others do not.
  - Some are ordered and others unordered.
- There are multiple types of collections
- Collection classes are available to us in Java's standard library of classes
- We import them into our projects using **`import`** statements
- In Java, Collections are in the **`java.util`** package

Packages and Imports

- Java has thousands of classes available to use and that's not even including third-party classes that are available for us to use.
- Packages provide a way to organize classes into logical groups.
- Specifying a package tells Java where to look for a class
  - This can also be used to specify which class you want to use when there are multiple classes with the same name
  - For instance: What if two third party classes **`Zapper`** existed? We'd need to specify which one we want to use.
    - **`com.techelevator.Zapper`** vs **`com.google.Zapper`**
- We tell our code what package a class lives in using the **`import`** statement.

Lists

- A List:
  - Is a collection of Objects of the same type
  - Is **Zero-indexed**, similar to arrays
  - Is an **ordered set of elements** accessible by index
  - **Allows duplicates**
  - **Can grow and shrink** as elements are added and removed
    - Methods: **`add()`** and **`remove()`**

- In Java, we use an **`ArrayList`** (object) to implement a **`List`** (interface)

Lists

- A List:
  - Is a collection of Objects of the same type
  - Is **Zero-indexed**, similar to arrays
  - Is an **ordered set of elements** accessible by index
  - **Allows duplicates**
  - **Can grow and shrink** as elements are added and removed
    - Methods: **`add()`** and **`remove()`**

- In Java, we use an **`ArrayList`** (object) to implement a **`List`** (interface)

Don't worry.. you aren't supposed to know what this means yet... it will make sense later. For now, just know the ArrayList is the most common implementation of a List.

What the Heck is List<T>?

List is defined as **`List<T>`**... what does this mean?

- **`List`** can be of any object type but we must specify the type
- The **`<T>`** is a placeholder that needs to be replaced with the type of object in the **`List`**. You will see **`<T>`** used any time a class can be used with different types of objects.

Creating A List

result is a `List` of `String` objects

Creating A List

result is a **`List`** of **`String`** objects

We are creating an **`ArrayList`** type of List which will contain **`String`** objects.

```
List<String> result = new ArrayList<String>();
```

Creating A List

result is a **`List`** of **`String`** objects

We are creating an **`ArrayList`** type of List which will contain **`String`** objects.

```
List<String> result = new ArrayList<String>();
```

We can omit the object type because it is specified in the **`List<String>`** declaration.

```
List<String> result = new ArrayList<>();
```

Primitive Wrapper Classes

- Lists and other collections can hold objects.
  - Wait… what if I want a list of ints? Or floats? Or doubles?

- Java has a wrapper class for each primitive data type.

Java Primitive Type	Wrapper Class	Constructor Argument
`byte`	`Byte`	`byte` or `String`
`short`	`Short`	`short` or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`float`	`Float`	`float`, `double`, or `String`
`double`	`Double`	`double` or `String`
`char`	`Character`	`char`
`boolean`	`Boolean`	`boolean` or `String`

Autoboxing and Unboxing

- **Autoboxing** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.
- **Unboxing** is converting an object of a wrapper type to its corresponding primitive value.
- The Java compiler applies unboxing when an object of a wrapper class is:
  - Passed as a parameter to a method that expects a value of the corresponding primitive type.
  - Assigned to a variable of the corresponding primitive type.

More info about Autoboxing/Unboxing can be found at:

https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html

Autoboxing Example

```
List<Integer> intList = new ArrayList<>();
for (int i = 1; i < 50; i++) {
    intList.add(i);
}
```

- Without autoboxing, the above code would result in a **compiler error** since we must add Objects to Lists, not primitive types.

- Autoboxing allows the java compiler to interpret the preceding code as the code below at compile time, “wrapping” the `int` in an `Integer` wrapper.

```
List<Integer> intList = new ArrayList<>();
for (int i = 1; i < 50; i++) {
    intList.add(Integer.valueOf(i));
}
```

Unboxing Example

```
public class Unboxing {

    public static void main(String[] args) {
        Integer i = new Integer(-8);

        // 1. Unboxing through method invocation
        int absVal = absoluteValue(i);
        System.out.println("absolute value of " + i + " = " + absVal);

        List<Double> ld = new ArrayList<>();
        ld.add(3.1416);    // # is autoboxed through method invocation.

        // 2. Unboxing through assignment
        double pi = ld.get(0);
        System.out.println("pi = " + pi);
    }

    public static int absoluteValue(int i) {
        return (i < 0) ? -i : i;
    }
}
```

For-Each Loops

- For-each is a loop specifically created to iterate through collections.
- Cannot modify contents of the collection used by the for-each loop during iteration.
- Useful to work with the elements when we don’t need to know index. When working with collections, the content is usually what is most useful, not the actual index position.

Queues

- **`Queues`** are a collection that provide additional functionality when adding and removing items.

- To add items, we **`offer()`**
  - **`offer`** returns a **`boolean`** to indicate success of adding an Object

- To remove items, we **`poll()`**
  - **`poll`** returns the next list item, or null if the queue is empty

- **`Queues`** operate as a **FIFO (First In, First Out)** structure

- **`Queues`** is an interface, so we instantiate a **`LinkedList`**

Stacks

- **`Stacks`** are a collection that provide additional functionality when adding and removing items.

- To add items, we **`push()`**
  - Pushes an item onto the top of a **`stack`**

- To remove items, we **`pop()`**
  - Removes the object at the top of the stack

- Stacks operate as a **LIFO (Last In, First Out)** structure