Arrays:

Arrays have a fixed length. You can't add or remove elements without shifting the other elements.

List Interface:

A list is any sequence of objects. Java classes that implement the ListInterface must have the methods of the ListInterface. Some of these include:

- **add**(**E** e)
  Appends the specified element to the end of this list (optional operation).
- **add**(int index, **E** element)
  Inserts the specified element at the specified position in this list (optional operation).
- **contains**(**Object** o)
  Returns true if this list contains the specified element.
- **set**(int index, **E** element)
  Replaces the element at the specified position in this list with the specified element (optional operation).
- **clear**()
  Removes all of the elements from this list (optional operation).

ArrayList:

ArrayLists work by having a backing array as their data structure. They implement the List Interface.

- That backing array has a certain length (capacity). If we need to add more to our ArrayList, but that would exceed the capacity, we create a new array with a larger length, copy all of the previous values into that new array, then set that new array to be the backing array for the ArrayList object.

Backing Array of the ArrayList (instantiating a blank ArrayList would lead to a constructor calling to make a new ArrayList with a `final int DEFAULT_CAPACITY`):

7	8	9	0			

Those blanks are set to 0, but they are not part of the array. The `.size()` method returns 4, while the backing array has slots for 7 elements.

If we add a few more elements:

7	8	9	0	67	-2	4

And then try to add a new value (say 100)

The backing array would need to be remade. Let's make a new array with double the size (which is what is typically done). An arraylist object might call a method like `reallocate()` to do this:

7	8	9	0	67	-2	4	100						

Now 100 can be added, as shown above.

If from here we wanted to add a new number (-375), it's no problem because the backing array is large enough to hold it. We could even add another number after that (12)
If we wanted to remove a number, like the one at index 2, that is also not a problem.

7	8	0	67	-2	4	100	-375	12					

Iterator Interface:

Iterators: an object that implements the **Iterator interface.**
The object can be used to traverse a `Collection` of objects.
It must start at the first index and go forward one index at a time. It cannot go backwards or jump around.

The Interface consists of 3 methods:

`boolean hasNext()//` returns true until there are no more elements to iterate

`Object next()` //returns an object reference and throws an `NoSuchElementException()` when there are no more items in the ArrayList

`void remove()`// can only call remove after next
has been called

If `remove()` is called before `next()`, then an `IllegalStateException` is thrown. You may need to make a boolean variable called `remove` or `legal` in the class implementing the interface to help you keep track of when to throw this exception.
Ex1:

```
String[] array = {"hi", "bye","ye","no"};
ArrayList<String> arr = new ArrayList<>(Arrays.asList(array));
Iterator iterate = arr.iterator();
while(iterate.hasNext()) { System.out.println(iterate.next()); }
```

An iterator object is set to iterate through the ArrayList arr. It will print out each element on its own line until there are no more elements left to print.

Ex2:

```
//predict the output of this one!
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(4);
        list.add(5);
        list.add(2);
        list.add(3);

        Iterator it = list.iterator(); //ArrayList class has the
iterator() method since it implements the Iterable interface

      it.next();
     System.out.println(it.hasNext());
     System.out.println(list);
      it.next();
      it.next();
      it.remove();
      System.out.println(list);
      System.out.println(it.next());
      it.next();
```

Output:

```
true
[4, 5, 2, 3]
[4, 5, 3]
3

Exception in thread "main" java.util.NoSuchElementException
      at java.base/java.util.ArrayList$Itr.next(ArrayList.java:970)
      at MyClass.main(MyClass.java:20)
```

Iterable Interface:

Allows for the use of a for-each loop.

It basically says “this Collection works with an iterator”.
It must implement the following:

```
public Iterator iterator(); //return an iterator for the object.
```

Intro to Generics:

You can make a class, like an ArrayList, hold a collection of objects of any type. You do this with <>. We’ve seen this before while making ArrayList.
It would be terribly inefficient if you need to make separate classes for IntArrayList, PersonArrayList, DoubleArrayList, PizzaArrayLists, etc.

Our class will take a **type parameter** - something in angle brackets.

We’ve seen this in ArrayList, and this is why the following code works just fine:

```
import java.util.*;
public class MyClass {
    public static void main(String args[]) {
        ArrayList<Integer> list = new ArrayList<Integer>();

        ArrayList<Calculator> list2 = new
ArrayList<Calculator>(Arrays.asList(new Calculator[]{new
Calculator(true, true), new Calculator(false,true)}));
    }
    private static class Calculator {
        boolean isTexasInstrument;
        boolean graphing;
        public Calculator(boolean texas, boolean graph) {
            isTexasInstrument = texas;
            graphing = graph;
        }
    }
}
```

We have a class called `Students`. `APStudents` is a subclass of `Students`. However, `ArrayList<APStudents>` is not a subclass of `ArrayList<Students>`. They are both just ArrayLists!

Making Generics:

When you make the class, put <E> next to the name of the class. It is customary to use E as the type parameter name, but you could use any variables.

Example:

```
static class Pair<K, V> {
     K first;
     V second;

     public Pair(K first_value, V second_value) {
          first = first_value;
          second = second_value;
     }
}
//main class:
Pair<Integer, String> p = new Pair(5, "hello");
System.out.println(p.first + " " + p.second);
```

Notice the brackets for type parameters get put on the class name, but not on the constructor.

Example 2:

If we wanted to make our own version of an ArrayList, and we needed to make the backing array, we could try this:

```
E[] data = (E[]) new Object[10];
```

Java does not let us make the array of type `E` directly, but since all classes are subclasses to `Object`, we can make an `Object` array and cast it.

If we need to get something from data, make sure to cast it (accessing an element from data returns an `Object`, not something of type `E`).

```
E favElement = (E) data[7];
```