English Name: ____________________

Java `ArrayList` Class

The `ArrayList` Class

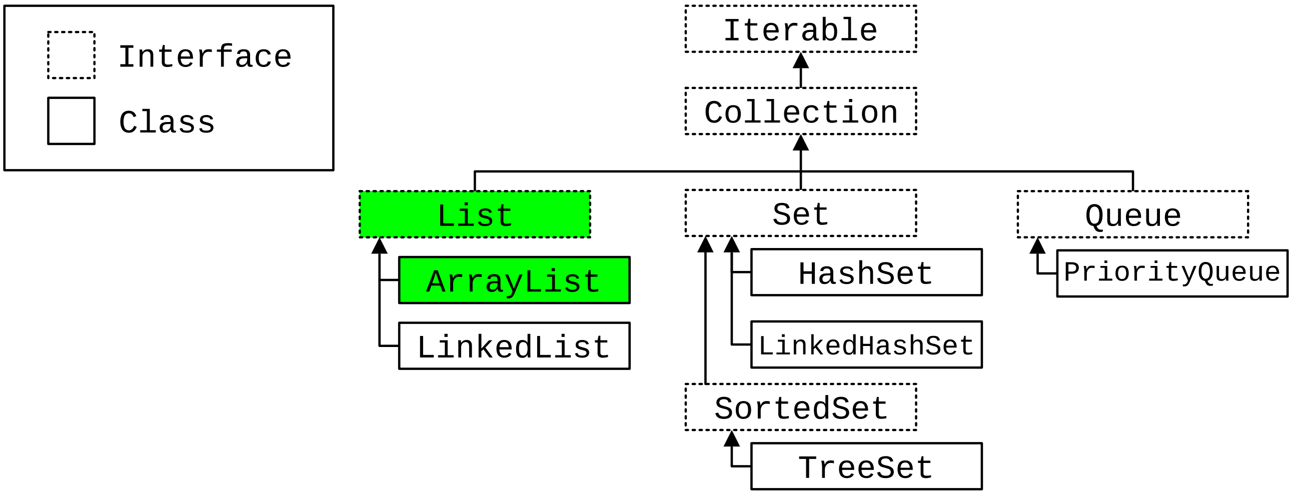
The `ArrayList` class implements a dynamic array. It provides methods to insert, delete, and retrieve values from an ordered list of elements, as well as other useful methods.

Arrays in Java have a fixed size, which means once you create an array, you cannot change it's length. If you wish to add a new element to, or delete an element from, an existing array, you will need to create a totally new array and copy elements from the original array into the new one. This requires extra code. The Java `ArrayList` class efficiently implements code with this functionality so that other programmers do not need to struggle to do so.

For *AP Computer Science A*, you will need to learn how to use the basic functionality of the `ArrayList` class, and be able to answer questions that require you to know the differences between the use of arrays versus the use of the `ArrayList` class.

Java Collections Hierarchy

The `ArrayList` class belongs to the *Collections Hierarchy*, a library of useful Java interfaces and utility classes that provide efficient data structures for storing, manipulating, and accessing groups of objects. Below is a diagram of some of the interfaces and classes within the Java *Collections Hierarchy*. The interface and class we discuss in this document are colored green.



The different interfaces and classes of the collections hierarchy are used for different purposes. For an ordered collection of objects that may contain duplicates, then an implementation of the `List` interface may be used. If order doesn't matter and duplicate elements are disallowed, then an implementation of the `Set` interface will likely be the better choice. If the elements must be ordered, then it may be most suitable to use the `PriorityQueue` class.

You will notice that there are different implementations of the `List` and `Set` interfaces shown. The reasons for choosing one implementation over the other is far beyond the scope *AP Computer Science A* and of this document. For students who aren't satisfied with that answer, a simplified explanation is: one implementation may be more efficient for certain uses – for example, an `ArrayList` may be more efficient when the collection is accessed (read) more frequently than modified, and a `LinkedList` may be more efficient when the collection elements will be added or deleted frequently. For most cases, the choice will be inconsequential.

The `List` Interface

The `List` interface defines all the methods required for an implementation of the `List` interface. In the diagram on the previous page, there are two different implementations of the `List` interface shown: `ArrayList` and `LinkedList`. If a program is implemented using an `ArrayList` and it is later found that a `LinkedList` would be more efficient, it can be relatively easy to change the underlying implementation with minimal changes to the program. (Although this does assume some competency by the original programmer.)

The methods included in the *AP Java Subset* that are applicable to the `ArrayList` class are all defined in the `List` interface (or inherited from the `Collection` interface). These are:

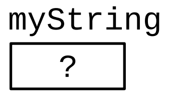
Method	**Description**
E **get**(int index)	Returns the element at position `index` in the list
boolean **add**(E obj) void **add**(int index, E obj)	If no `index` is given, the element, `obj`, is appended to the end of the list and `true` is returned. If an index is given, the element, `obj`, is inserted into the list at the index given by `index`, shifting all elements starting from that index one position to the right.
E **set**(int index, E obj)	Replaces the element at position `index` with the element `obj`; returns the element that was previously at that position.
E **remove**(int index)	Removes the element from position `index`, shifting all elements after that element to the left by one index position.
int **size**()	Returns the number of elements in the list.

You can build a strong understanding of the `ArrayList` class by completing the guided exercises.

Declaring and Initializing an Object (Review)

In this section, we review how to declare and initialize objects, taking a `String` object as an example. The following declaration of a `String` object allocates a place to store the reference to a `String` object. The result of the statement is shown diagrammatically to the right of the statement.

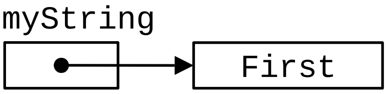
```
String myString;
```



It is important to understand that the statement only creates a place to store the reference to the `String`, it does not allocate any space to store a `String` object. Also, the variable is not initialized to any particular value (represented by the question mark in the diagram), so trying to read the value will result in an error.

If we wish to actually instantiate a `String` object, or any other object, we must call the constructor for the class. The constructor is a special method within the class that shares the same name as the class and has the purpose of initializing the fields of the class. The following statement declares and initializes a `String` object. Again, the result of the statement is shown diagrammatically to the right of the statement.

```
String myString = new String("First");
```



Java Generics and Declaring an ArrayList

When we declare an array of to store data, we specify the type of element to be stored within the array, followed by square brackets. For example: `String[] myStrArray` declares an array that may store objects of type `String` (or more accurately references to objects of type `String`).

When we declare an `ArrayList` object, the type is `ArrayList`. Yet we need a way to say what type of object will be stored as elements within the `ArrayList`. This is done using ***generics***, also known as *parameterized types*.

Recall that the values passed to a method are called the method ***parameters***, and the parameters are enclosed in parentheses immediately after the method name. For example, the `String` class has a method named `substring` that takes two parameters of type `int`, named `from` and `to` in the method signature.

$$\texttt{String substring(}\underbrace{\texttt{int from, int to}}_{\textit{parameters}}\texttt{)}$$

When we have a class that requires a ***type parameter***, the *type parameter* is placed within angle brackets (between < and >). As examples, an `ArrayList` requires a *type parameter* to know what type of objects the `ArrayList` will be used to store. The following statement declares an `ArrayList` named `myStrList` that can store objects of type `String`.

```
ArrayList<String> myStrList;
```

If more than one type parameter is required, such as with the `Map` class, the type parameters are separated by commas: `Map<String, Integer> myMap;` .

Initializing an `ArrayList`

To instantiate an object of any class, the ***constructor*** is called. The following statement declares variable of type `ArrayList` that will store objects of type `String`. The statement passes the type `String` as the type parameter. It then instantiates an object of type `ArrayList`, again passing the type `String` to the constructor as the type parameter.

$$\underbrace{\texttt{ArrayList}}_{\textit{data type}}\texttt{<}\underbrace{\texttt{String}}_{\substack{\textit{type}\\\textit{parameter}}}\texttt{> }\underbrace{\texttt{myStrList}}_{\substack{\textit{reference}\\\textit{variable}}}\texttt{ = new }\underbrace{\texttt{ArrayList}}_{\textit{constructor}}\texttt{<}\underbrace{\texttt{String}}_{\substack{\textit{type}\\\textit{parameter}}}\texttt{>}\underset{\uparrow\ \substack{\textit{no method}\\\textit{parameters}}}{\texttt{()}}\texttt{;}$$

The type parameter in the data type (on the left) must be the same as the type parameter passed to the constructor (on the right). In this example, the `ArrayList` constructor takes a type parameter but takes no method parameters – the parentheses are empty.

Java `ArrayList` Class

Java *generics* (type parameters) were introduced in Java 5, and the type parameter was required until the “*diamond operator*” was introduced in Java 7. Now, the angle brackets after the constructor can be left empty and the type will be inferred from the type parameter passed to the data type. Thus, the statement below is equivalent to the previous Java statement, with the opening and closing angle brackets (<>) being referred to as the *diamond operator*.

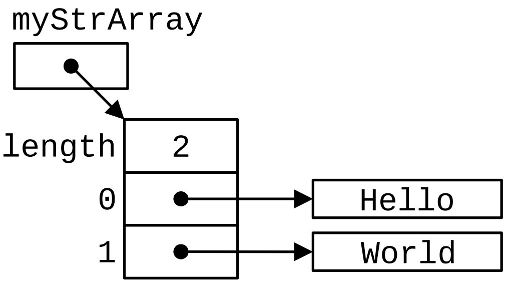
$$\texttt{ArrayList<}\underbrace{\texttt{String}}_{\textit{type parameter}}\texttt{> myStrList = new }\underbrace{\texttt{ArrayList}}_{\textit{constructor}}\underbrace{\texttt{<>}}_{\textit{diamond operator}}\texttt{();}$$

Using `ArrayList`

Consider the following code and the diagram that represents the structure created in memory.

```
1  String[] myStrArray = new String[2];
2  myStrArray[0] = "Hello";
3  myStrArray[1] = "World";
```

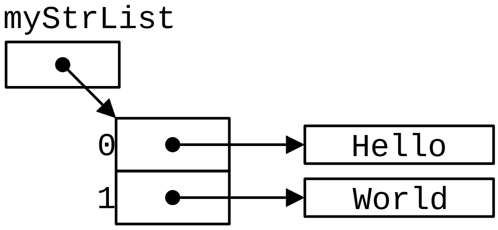
Line `1` declares and initializes an empty array that can store two elements, while lines `2` and `3` set the references at array index `0` and array index `1` to the `String` values “`Hello`” and “`World`”, respectively.



The following code creates approximately the same data structure, implemented using an `ArrayList` rather than an array.

```
1  ArrayList<String> myStrList = new ArrayList<>();
2  myStrList.add("Hello");
3  myStrList.add("World");
```

The diagram to the right represents the structure the above code creates in memory. Line `1` of the code declares and initializes an empty `ArrayList` that can store an indefinite number of elements (limited by the amount of memory the program can use). Lines `2` and `3` add the `String` values “`Hello`” and “`World`”, each time to the end of the list.



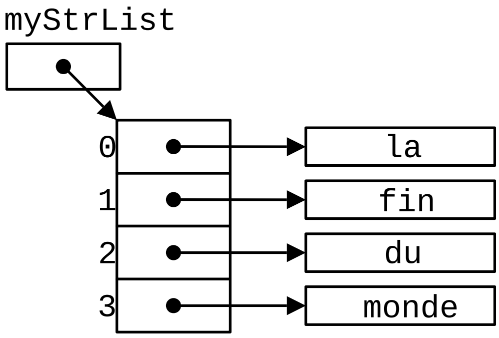
The remaining methods in the *Java AP Subset* are described in a table at the end of this document. Their use will be learned by completing exercises.

When to Use Arrays

Comparing `ArrayList` to arrays, it may seem that `ArrayList`, in almost all cases, is more convenient to use than `ArrayList`. Here we discuss two particular use cases when we may choose to use an array rather than an `ArrayList`: firstly, when the data is unchanging, and secondly, when there is a lot of data that may be stored as a primitive type.

Use Arrays for Unchanging Data

If the number of data items to be stored is unchanging and known at the time of writing the program, then initializing an array is slightly more compact and convenient, and the additional functionality of an `ArrayList` will not be used. Consider the following data structure and the code for creating the structure as an array versus as an `ArrayList`.



Array of `String` ***Implementation***

```java
String[] myStrArray = { "la", "fin", "du", "monde" };
```

`ArrayList` ***of*** `String` ***Implementation***

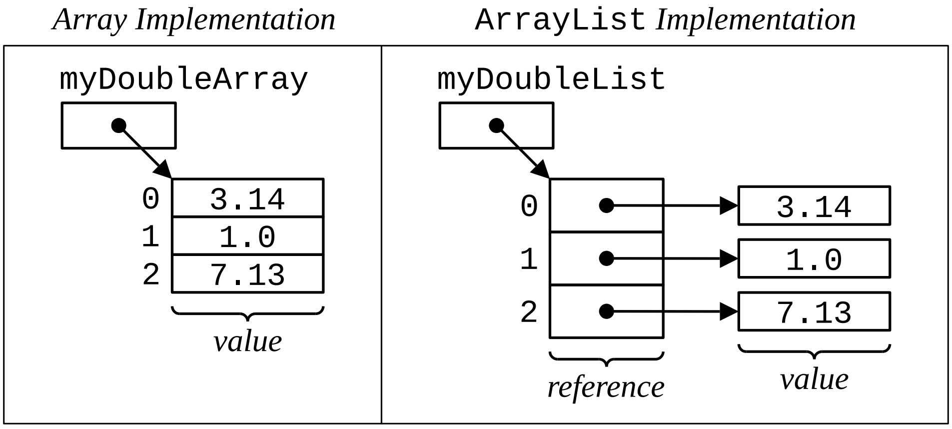
```java
ArrayList<String> myStrList = new ArrayList<>();
myStrList.add("la");
myStrList.add("fin");
myStrList.add("du");
myStrList.add("monde");
```

The array implementation does not require method calls, and the compiler is able create the data structure. For the `ArrayList` implementation, creating the data structure requires method calls and the execution of code. This will be done at run time, so likely adds some small overhead in execution time and memory use. In almost every way an array is preferable to an `ArrayList`.

Consider Using an Array for Storing Large Amounts of Primitive Type Data

The *type parameter* passed to `ArrayList` must be an object. This means the type parameter cannot be a *primitive type*.

In order to store primitive types – such as `int`, `double`, or `boolean` – in an `ArrayList`, objects of the appropriate wrapper class (`Integer`, `Double`, or `Boolean`) must be used. The diagram below compares the data structures created to store a few `double` types in an array versus storing the same values in an `ArrayList`, and example code that will create each structure is given below the diagram.



Array of* `double` *Implementation

```
double[] myDoubleArray = { 3.14, 1.0, 7.13 };
```

`ArrayList` *of* `Double` *Implementation*

```
ArrayList<Double> myDoubleList = new ArrayList<>();
myStrList.add(3.14);
myStrList.add(1.0);
myStrList.add(7.13);
```

In terms of memory space, the array implementation need only store the data values, whereas the `ArrayList` implementation needs to store, at minimum, the data values and the references (locations where to find the reference values).

In terms of execution time, to access a data item in an array, the computer need only calculate the offset from the start of the array and read the data value; whereas to access a data item in an ArrayList, reading the offset from the start of the array results in the reference value, and the computer is then required to perform another read to get the value that is referred to by the reference. (Memory fetching and caching is another, more advanced consideration: the values stored in an array are contiguous, whereas values stored in an `ArrayList` of objects will likely not be).

Using an `ArrayList` for primitive types will be much less efficient than using an array. However, with the speed and memory capacity of modern computers, choosing an array over an `ArrayList` should only impact performance when dealing with a very large amount of data or when the data is used in a performance-critical section.

Java AP Subset `ArrayList` Methods

The following table provides statements that show the use of common methods that modify an `ArrayList` of `String`, along with equivalent statements for an array of `String`.

`int i = myStrList.size();`	`int i = myStrArray.length;`
`myStrList.add("World");` `myStrList.add(0, "Hello");`	There are no array equivalents; this method appends a new element to the array, or inserts an element at the given index.
`myStrList.set(1, "teacher");`	`myStrArray[1] = "teacher";`
`String s = myStrList.get(1);`	`String s = myStrArray[1];`
`MyStrList.remove(1);`	There is no array equivalent; this method removes an element from the list.