COMPSCI 121: ARRAYLIST, WRAPPER CLASSES, REFERENCE VARIABLES

SPRING 2020

WHAT SHOULD YOU DO IF YOU NEED HELP?

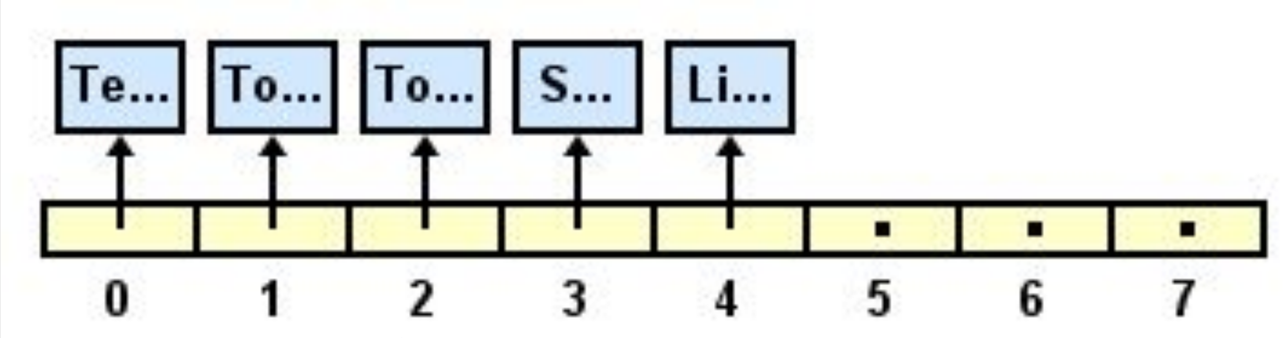
- Visit the professors and TAs during Zoom office hours- links on Moodle.

- Ask questions in Moodle private forum or Piazza (public / private posts).

GOALS FOR TODAY'S CLASS

- ArrayLists
- Wrapper Classes
- Parameters of reference types

THE PROBLEM WITH ARRAYS

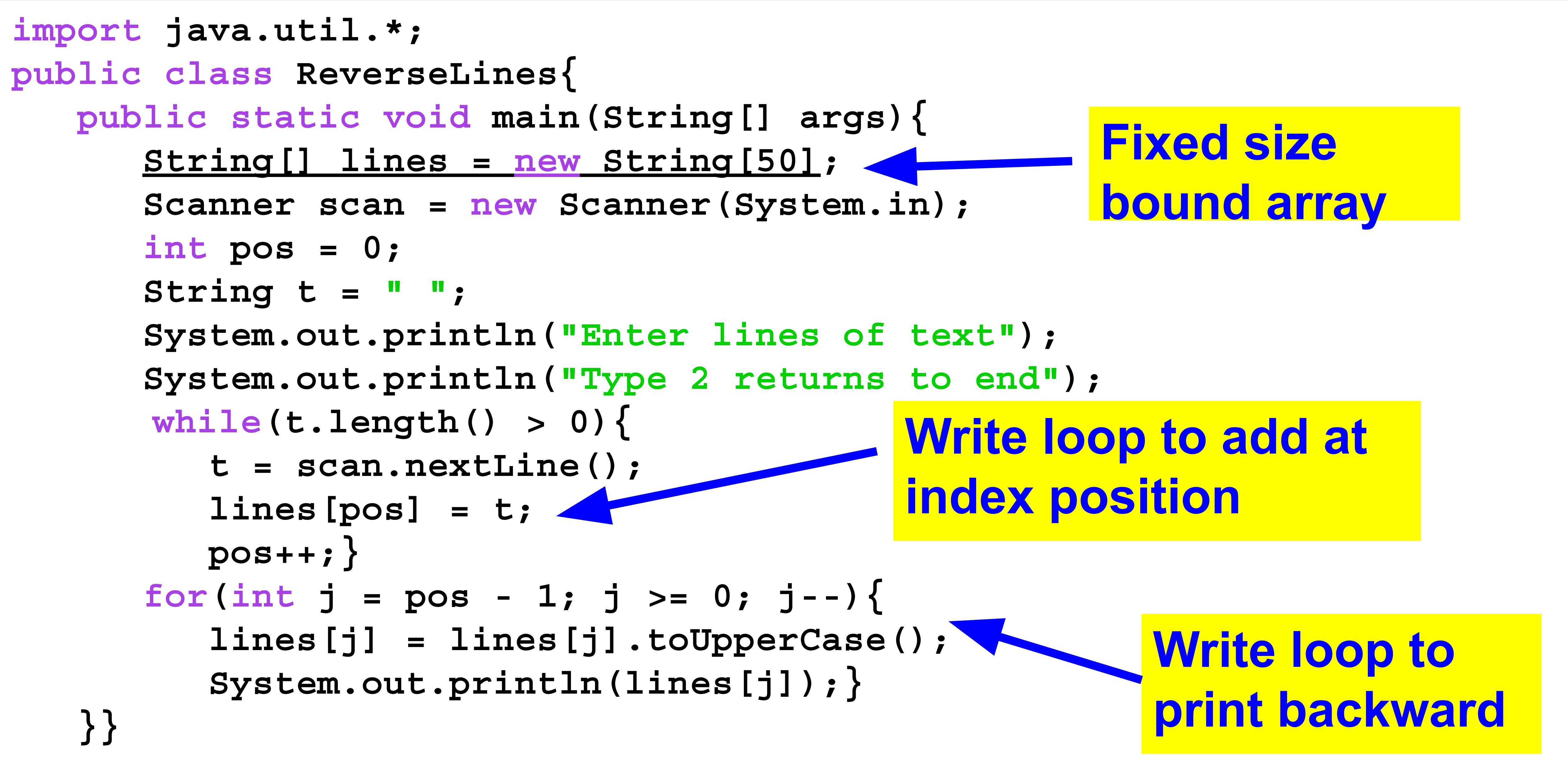


Cells with data- objects

Empty cells- contain `null`

- **Arrays are fixed size- inconvenient if you don’t know how much data you will encounter.**
- **Have to write code for adding, deleting, traversing, expanding arrays.**
- **Have to take care of NullPointerExceptions.**

EXAMPLE 1: THE PROBLEM WITH ARRAYS



```
import java.util.*;
public class ReverseLines{
   public static void main(String[] args){
      String[] lines = new String[50];
      Scanner scan = new Scanner(System.in);
      int pos = 0;
      String t = " ";
      System.out.println("Enter lines of text");
      System.out.println("Type 2 returns to end");
      while(t.length() > 0){
         t = scan.nextLine();
         lines[pos] = t;
         pos++;}
      for(int j = pos - 1; j >= 0; j--){
         lines[j] = lines[j].toUpperCase();
         System.out.println(lines[j]);}
   }}
```

Fixed size bound array

Write loop to add at index position

Write loop to print backward

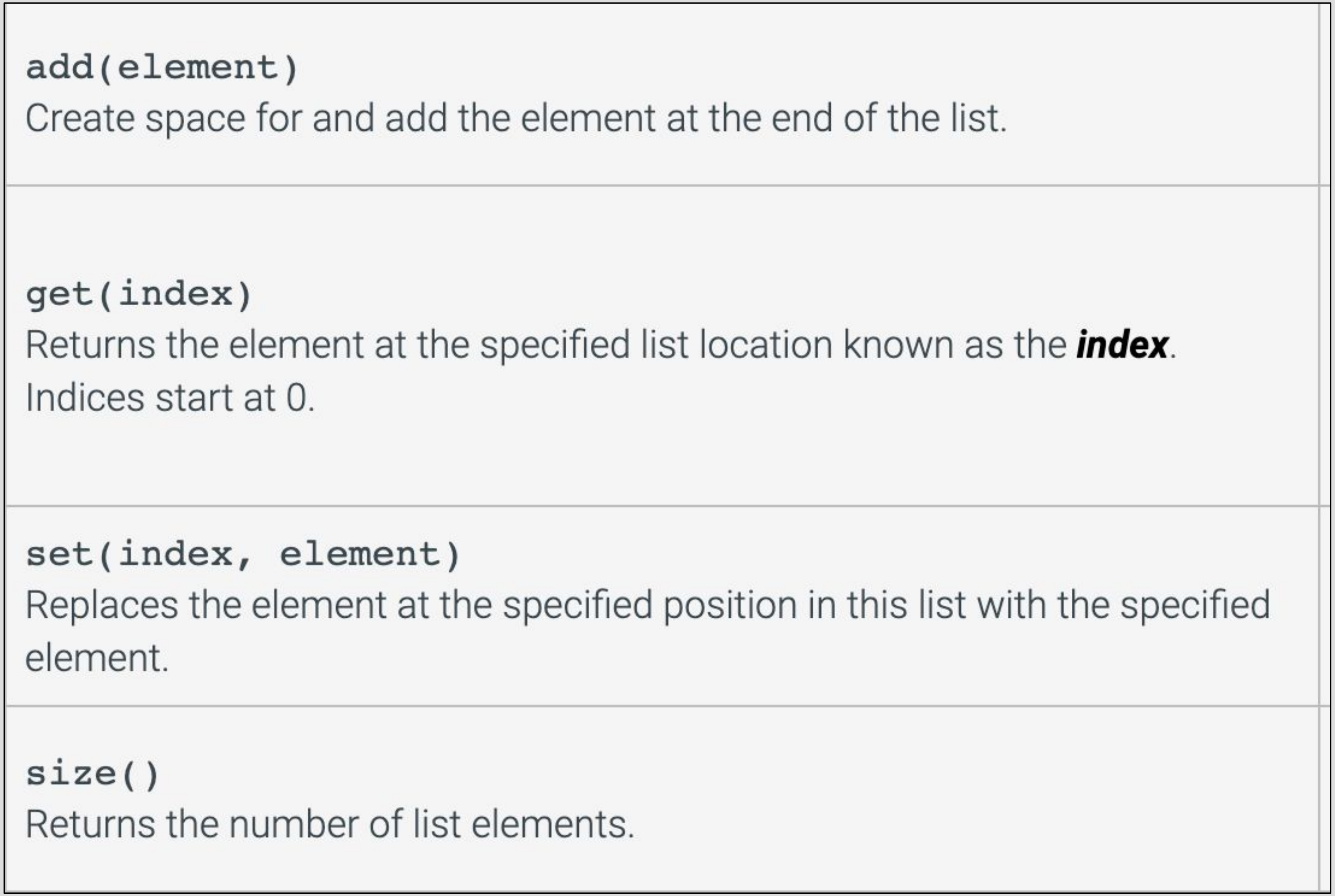
THE ARRAYLIST CLASS

ArrayList: stores data in an array, but:

- no need to specify an initial size.
- no need to use index numbers to add, remove.
- automatically resizes the array as needed.
- encapsulates an array that can store any Object.

User needs only to know about interface (ArrayList public methods, or API) and NOT the implementation (the indexing details of adding, removing, etc.).

ARRAYLIST API



`add(element)`
Create space for and add the element at the end of the list.

`get(index)`
Returns the element at the specified list location known as the ***index***.
Indices start at 0.

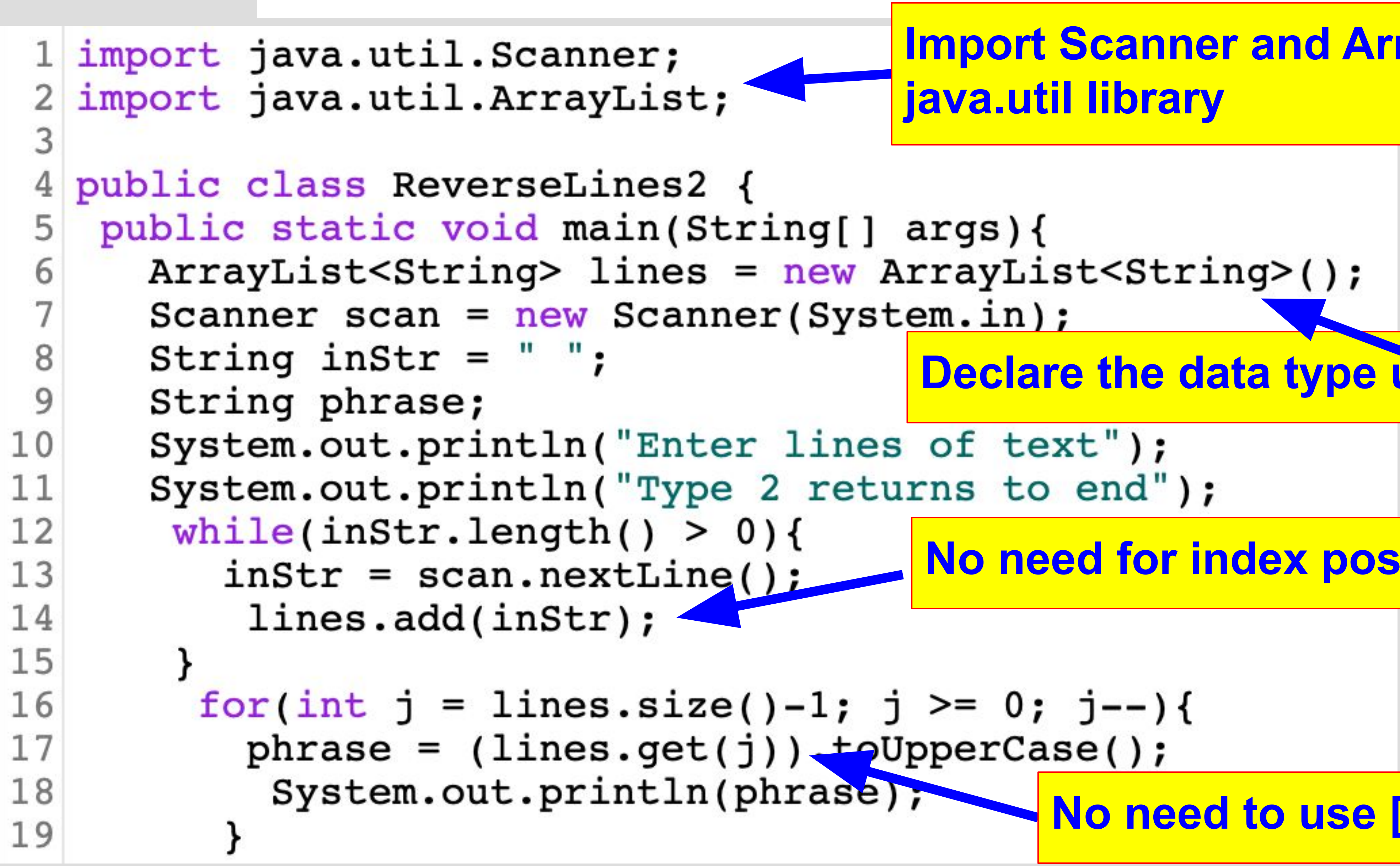
`set(index, element)`
Replaces the element at the specified position in this list with the specified element.

`size()`
Returns the number of list elements.

See Java API

Method summary.

EXAMPLE 1: WITH ARRAYLIST



```
import java.util.Scanner;
import java.util.ArrayList;

public class ReverseLines2 {
 public static void main(String[] args){
   ArrayList<String> lines = new ArrayList<String>();
   Scanner scan = new Scanner(System.in);
   String inStr = " ";
   String phrase;
   System.out.println("Enter lines of text");
   System.out.println("Type 2 returns to end");
    while(inStr.length() > 0){
      inStr = scan.nextLine();
       lines.add(inStr);
    }
     for(int j = lines.size()-1; j >= 0; j--){
       phrase = (lines.get(j)).toUpperCase();
        System.out.println(phrase);
      }
```

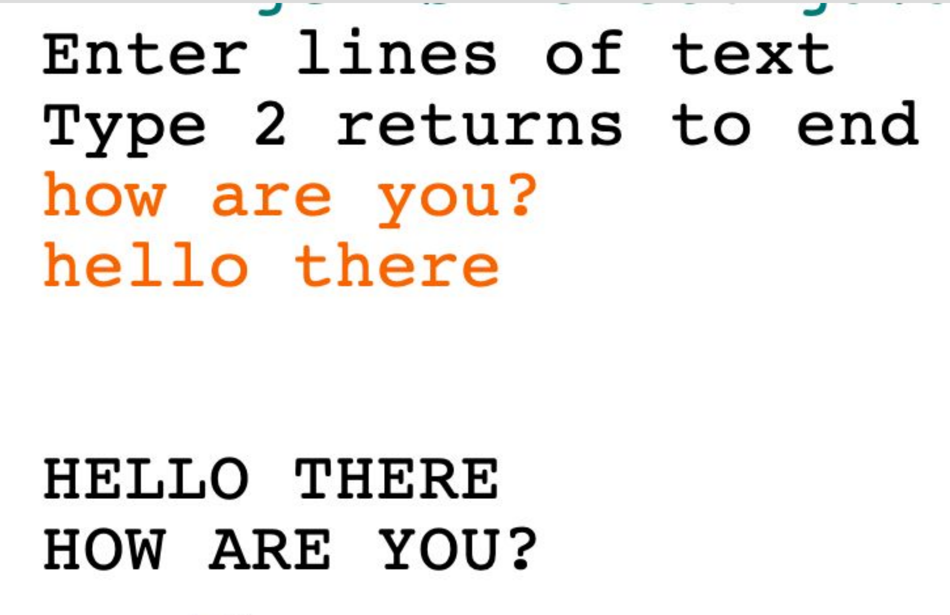
Import Scanner and ArrayList from java.util library

Declare the data type using < >

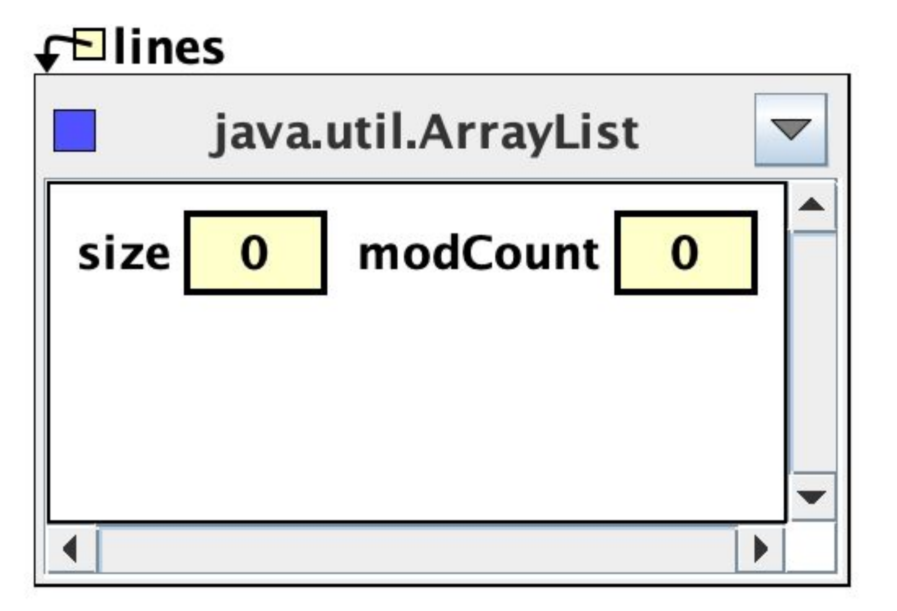
No need for index position

No need to use [ ]

EXAMPLE 1: WITH ARRAYLIST



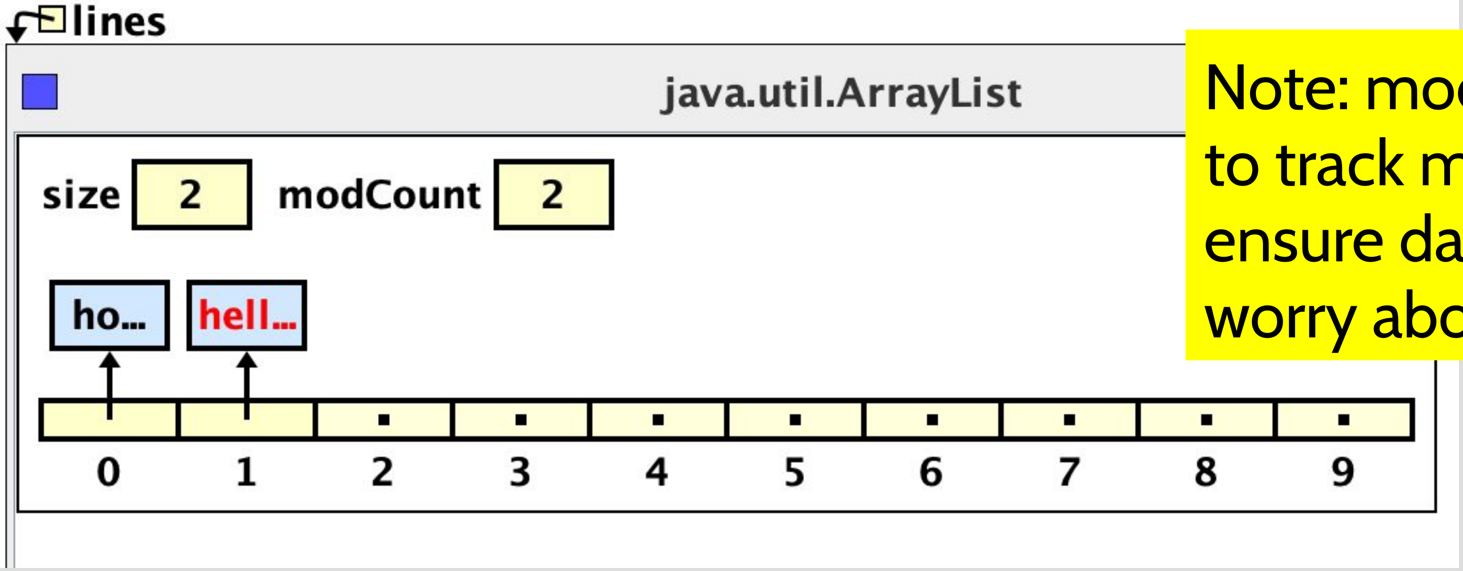
```
Enter lines of text
Type 2 returns to end
how are you?
hello there



HELLO THERE
HOW ARE YOU?
```



lines

java.util.ArrayList

size 0 modCount 0



lines

java.util.ArrayList

size 2 modCount 2

Note: modCount is used internally to track modifications to the list to ensure data is correct. No need to worry about it.

EXAMPLE: ARRAYLIST WITH STRINGS

```
import java.util.ArrayList;
public class FruitArrayList {
  public static void main(String args[]) {

 ArrayList<String>fruits = new ArrayList<String>();

 /*This is how elements should be added to the array list*/
 fruits.add("Apple");
 fruits.add("Mango");

 /* Displaying array list elements */
 System.out.println("Current arrayList is:"+ fruits +" and
the size is "+ fruits.size());
```

Note import statement

String type

adds to end of ArrayList

size method instead of length

```
Current arrayList is:[Apple, Mango] and the size is 2
```

EXAMPLE ARRAYLIST ADD and REMOVE

ArrayList state from previous slide: [Apple, Mango]

```
fruits.add(0,"Cherry");
```

[Cherry, Apple, Mango]

```
fruits.add(1,"Strawberry");
```

[Cherry, Strawberry, Apple, Mango]

```
fruits.remove("Apple");
```

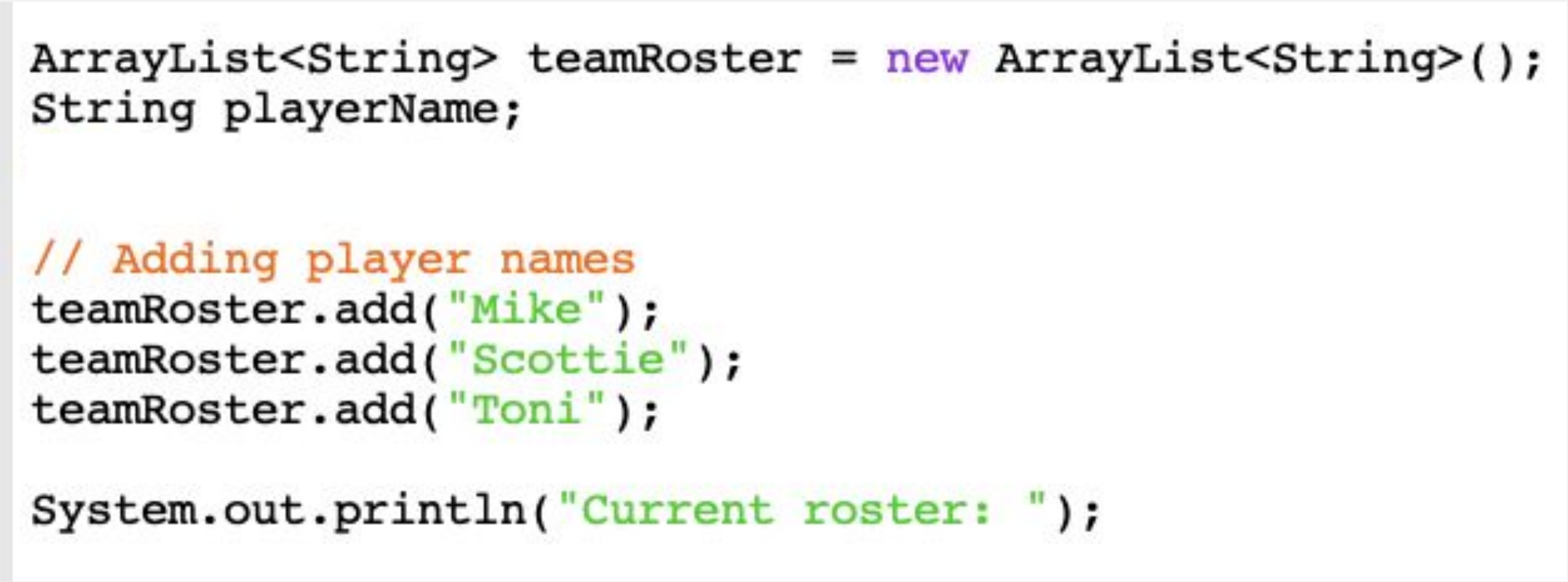
[Cherry, Strawberry, Mango]

```
fruits.remove(1);
```

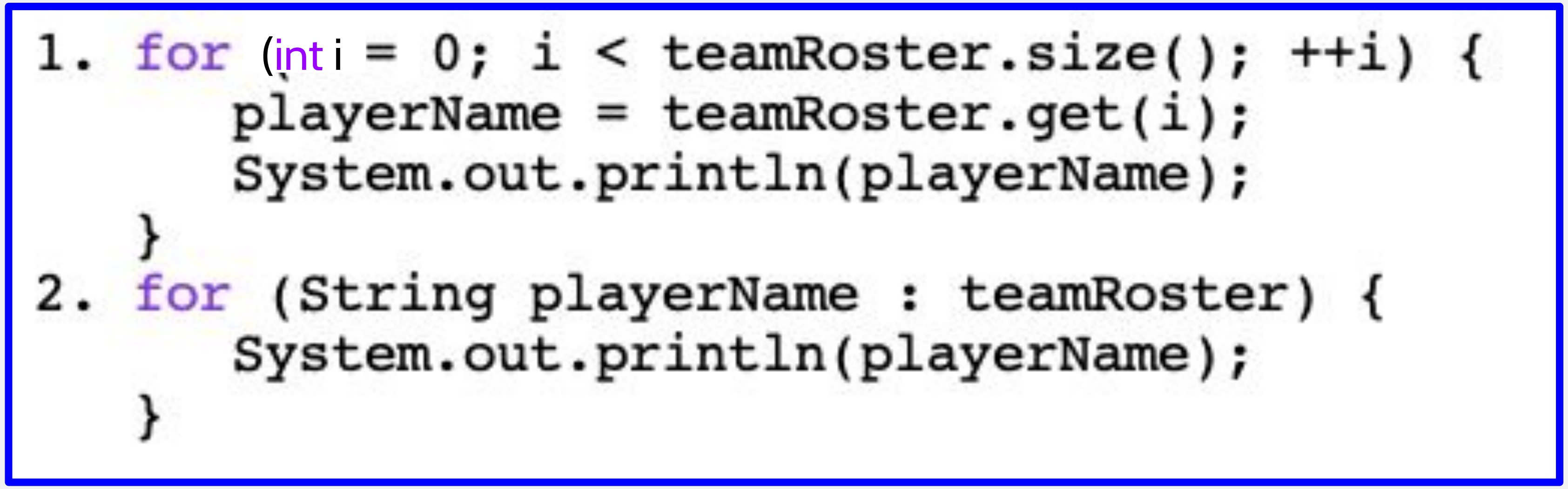
[Cherry, Mango]

NOTE: data is kept contiguous after removal.

CLICKER QUESTION 1



```
ArrayList<String> teamRoster = new ArrayList<String>();
String playerName;


// Adding player names
teamRoster.add("Mike");
teamRoster.add("Scottie");
teamRoster.add("Toni");

System.out.println("Current roster: ");
```



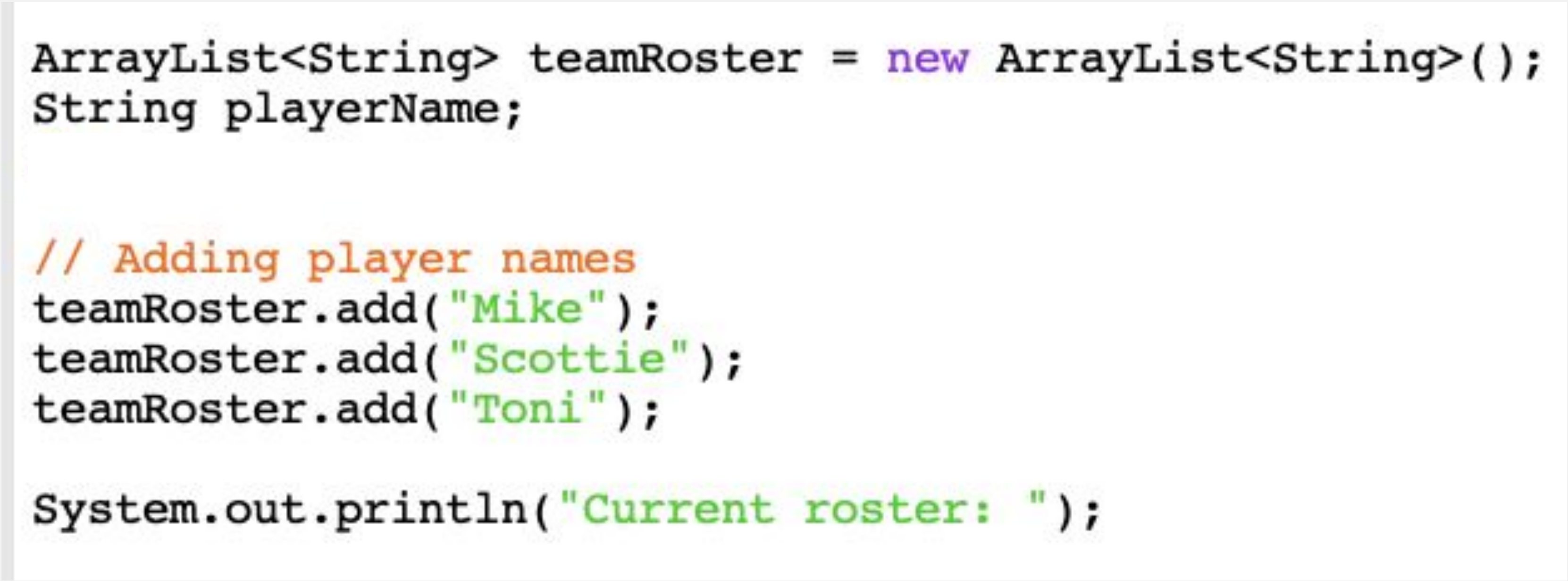
```
1. for (int i = 0; i < teamRoster.size(); ++i) {
       playerName = teamRoster.get(i);
       System.out.println(playerName);
   }
2. for (String playerName : teamRoster) {
       System.out.println(playerName);
   }
```

Which of the loops would correctly print out the names of the players?

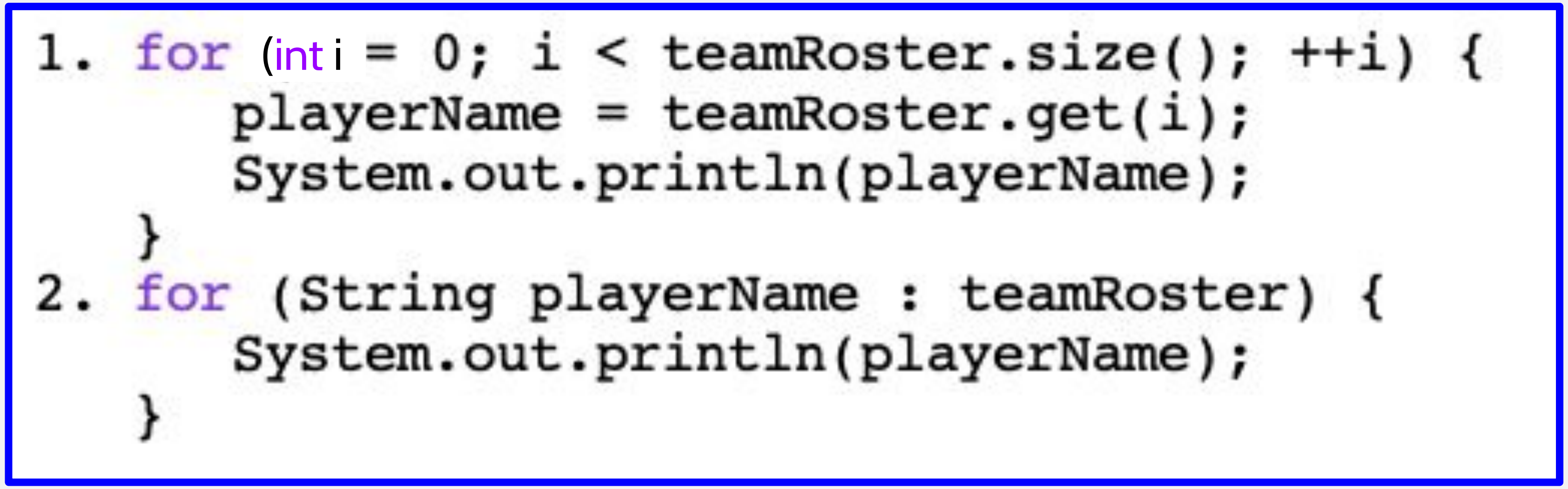
A. 1
B. 2.
C. 1 and 2
D. None

Ready for Answer 1?

CLICKER QUESTION 1



```
ArrayList<String> teamRoster = new ArrayList<String>();
String playerName;


// Adding player names
teamRoster.add("Mike");
teamRoster.add("Scottie");
teamRoster.add("Toni");

System.out.println("Current roster: ");
```



```
1. for (int i = 0; i < teamRoster.size(); ++i) {
      playerName = teamRoster.get(i);
      System.out.println(playerName);
   }
2. for (String playerName : teamRoster) {
      System.out.println(playerName);
   }
```

Which of the loops would correctly print out the names of the players?

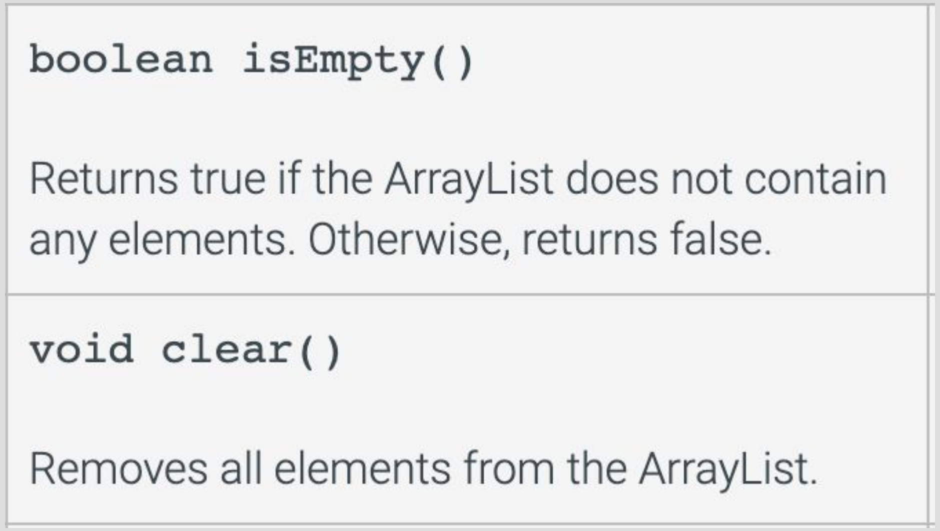
A. 1

B. 2.

C. 1 and 2

D. None

OTHER ARRAYLIST METHODS



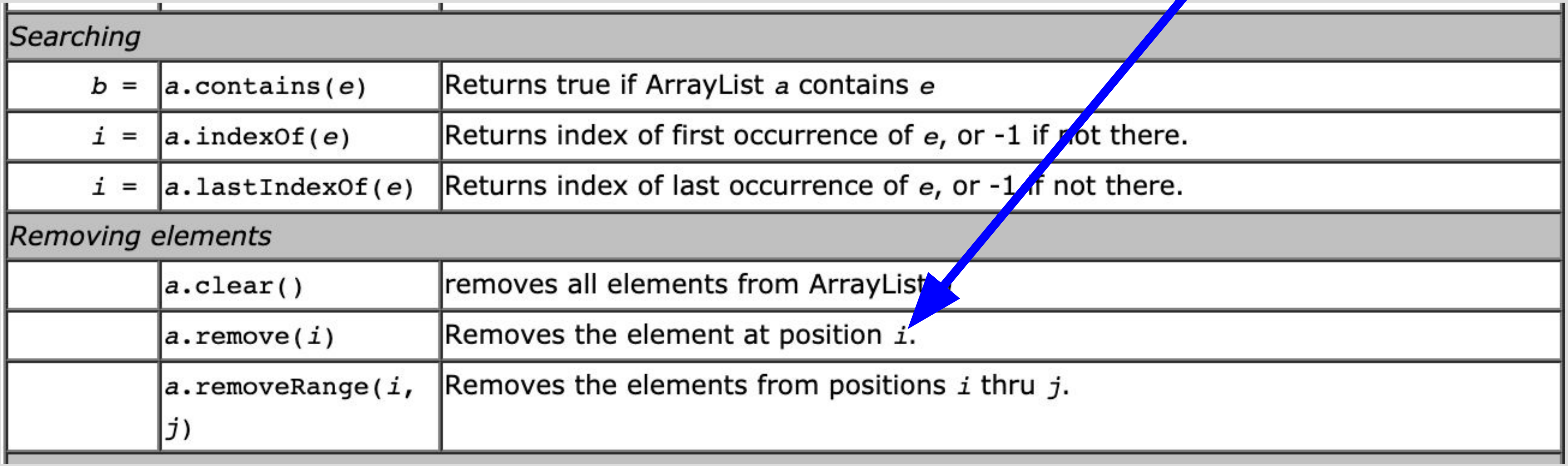
`boolean isEmpty()`

Returns true if the ArrayList does not contain any elements. Otherwise, returns false.

`void clear()`

Removes all elements from the ArrayList.

You still have to be careful- an `indexOutOfBoundsException` will be thrown if you pass in an invalid index!



Searching		
b =	`a.contains(e)`	Returns true if ArrayList *a* contains *e*
i =	`a.indexOf(e)`	Returns index of first occurrence of *e*, or -1 if not there.
i =	`a.lastIndexOf(e)`	Returns index of last occurrence of *e*, or -1 if not there.
Removing elements		
	`a.clear()`	removes all elements from ArrayList *a*
	`a.remove(i)`	Removes the element at position *i*.
	`a.removeRange(i, j)`	Removes the elements from positions *i* thru *j*.

WRAPPER CLASSES

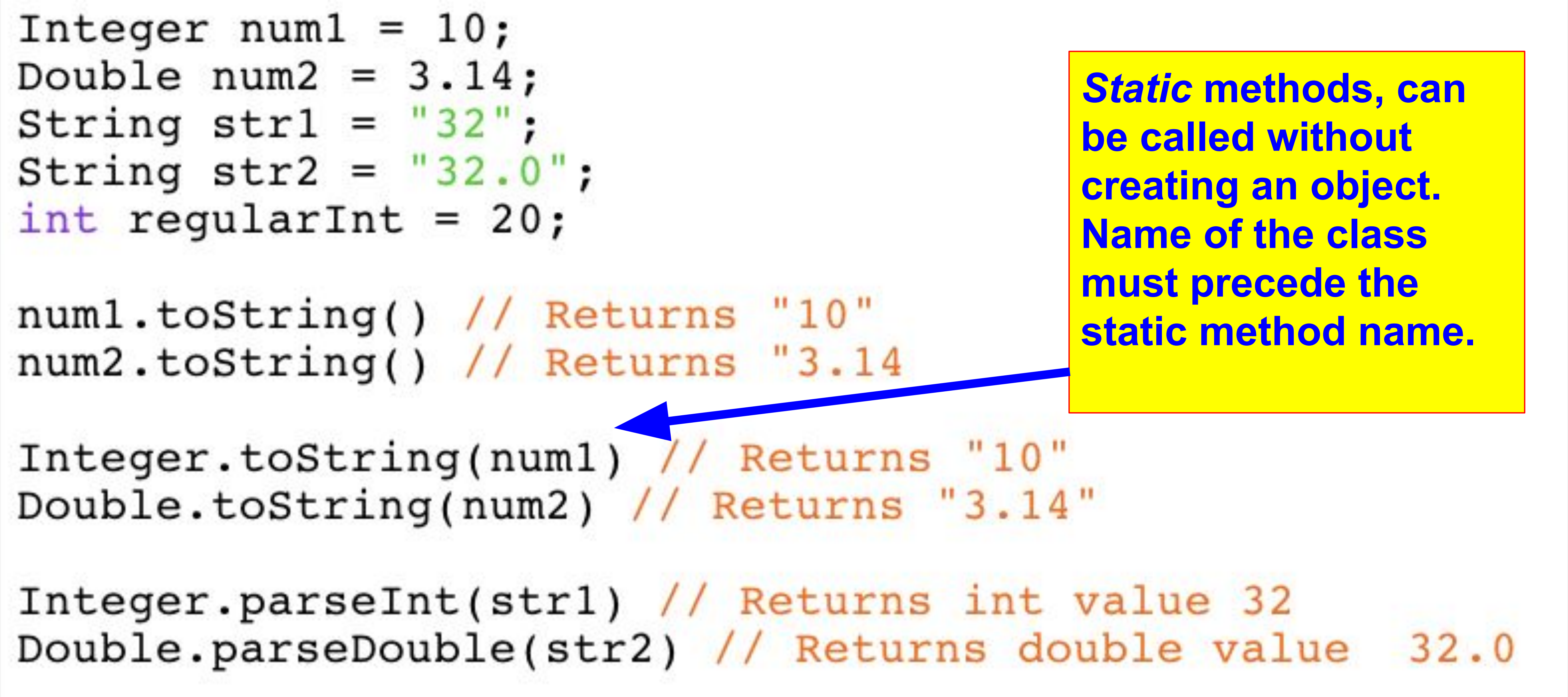
Primitive data types can be handled as Objects by using Wrapper Classes.

- `Integer`, the wrapper for `int`
- `Double`, the wrapper for `double`
- `Character`, the wrapper for `char`
- `Boolean` the wrapper for `boolean`

This allows you to call useful methods to work with primitive data, such as converting to an int from a String:

```
int num = Integer.parseInt(“1005”);
String numStr = Integer.toString(num);
```

WRAPPER CLASS METHODS TO CONVERT TO AND FROM STRINGS



```
Integer num1 = 10;
Double num2 = 3.14;
String str1 = "32";
String str2 = "32.0";
int regularInt = 20;

num1.toString() // Returns "10"
num2.toString() // Returns "3.14

Integer.toString(num1) // Returns "10"
Double.toString(num2) // Returns "3.14"

Integer.parseInt(str1) // Returns int value 32
Double.parseDouble(str2) // Returns double value  32.0
```

***Static* methods, can be called without creating an object. Name of the class must precede the static method name.**

WRAPPER CLASSES

ArrayLists *only work* with class types, not primitive types. You will need wrapper class for storing them.

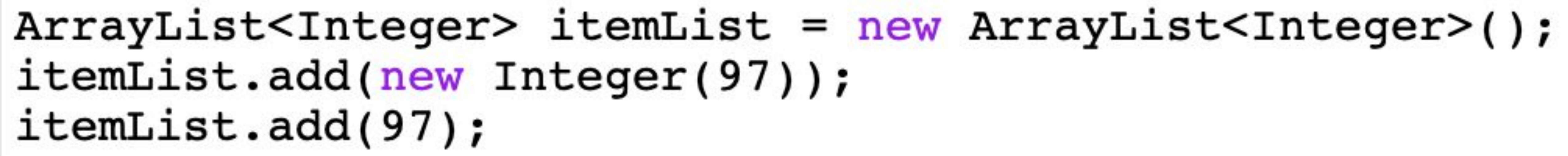
- `Integer`, the wrapper for `int`
- `Double`, the wrapper for `double`
- `Character`, the wrapper for `char`
- `Boolean` the wrapper for `boolean`

`ArrayList` with wrapper types: `<Integer>, <String>, <Boolean>`.....

Note: automatically converts primitives to Object through autoboxing:

```
list.add(100);
```

ARRAYLISTS WITH WRAPPER CLASSES



```
ArrayList<Integer> itemList = new ArrayList<Integer>();
itemList.add(new Integer(97));
itemList.add(97);
```

This is an “autoboxing” call- Java will create and add a new Integer(97)

Both add calls will result in a new Integer object that contains 97 being added to the integerList.

CLICKER QUESTION 2

Which of the following creates an `ArrayList` that can store these data?

100, 4, -27, 30

```
A. ArrayList<Integer> intList = new
   ArrayList<Integer>();
B. ArrayList<int> intList = new ArrayList<int>();
C. ArrayList<Integer> intList = new
   ArrayList<int>();
D. ArrayList<int> intList = new
   ArrayList<Integer>();
```

Ready for Answer 2?

CLICKER QUESTION 2 ANSWER

Which of the following creates an ArrayList that can store these data?

100, 4, -27, 30

A. `ArrayList<Integer> intList = new ArrayList<Integer>();`

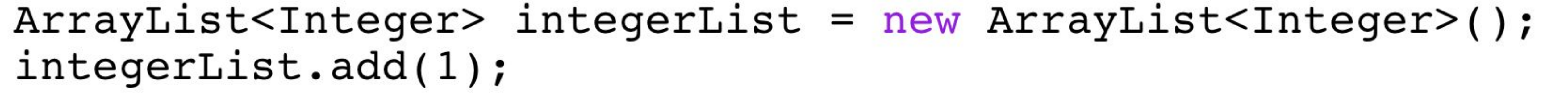
B. `ArrayList<int> intList = new ArrayList<int>();`

C. `ArrayList<Integer> intList = new ArrayList<int>();`

D. `ArrayList<int> intList = new ArrayList<Integer>();`

primitive types not allowed

CLICKER QUESTION 3



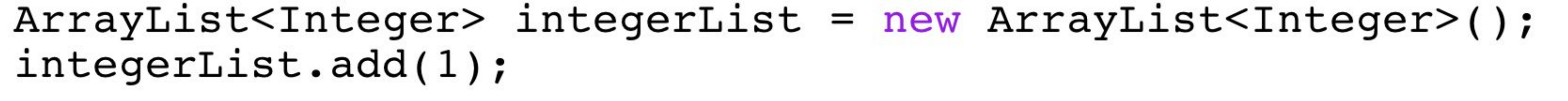
```java
ArrayList<Integer> integerList = new ArrayList<Integer>();
integerList.add(1);
```

Which of the following statements is true of the code above?

A. The ArrayList `integerList` has length 1.
B. Autoboxing converts `int` to `Integer` in `add`.
C. The `add` method will fail.
D. The add method adds the `int` value to position 1.

Ready for Answer 3?

CLICKER QUESTION 3 ANSWER



```
ArrayList<Integer> integerList = new ArrayList<Integer>();
integerList.add(1);
```

Autoboxing automatically converts primitives to Object.

Which of the following statements is true of the code above?

A. The ArrayList `integerList` has length 1. `//Default 10`

B. Autoboxing converts `int` to `Integer` in `add`.

C. The `add` method will fail. `//No error`

D. The add method adds the int value to position 1 `//at index 0`

VARIABLES OF PRIMITIVE TYPES AND METHODS

Passing a variable that references a ***primitive type*** to a method results in ***no change*** to the value. Consider the changeAge method:

```
public void changeAge(int ageParam) {
      ageParam = 110;
 }
```

The value of the variable “age” did not change. That’s because the parameter is a copy and the copy exists only in the scope of the method.

```
int age = 20;
System.out.println(age);  // prints 20
changeAge(age);
System.out.println(age);  // prints 20
```

VARIABLES OF REFERENCE TYPES AND METHODS

Passing a variable that references an Object to a method means that that Object ***can be modified*** in the method. Consider this method:

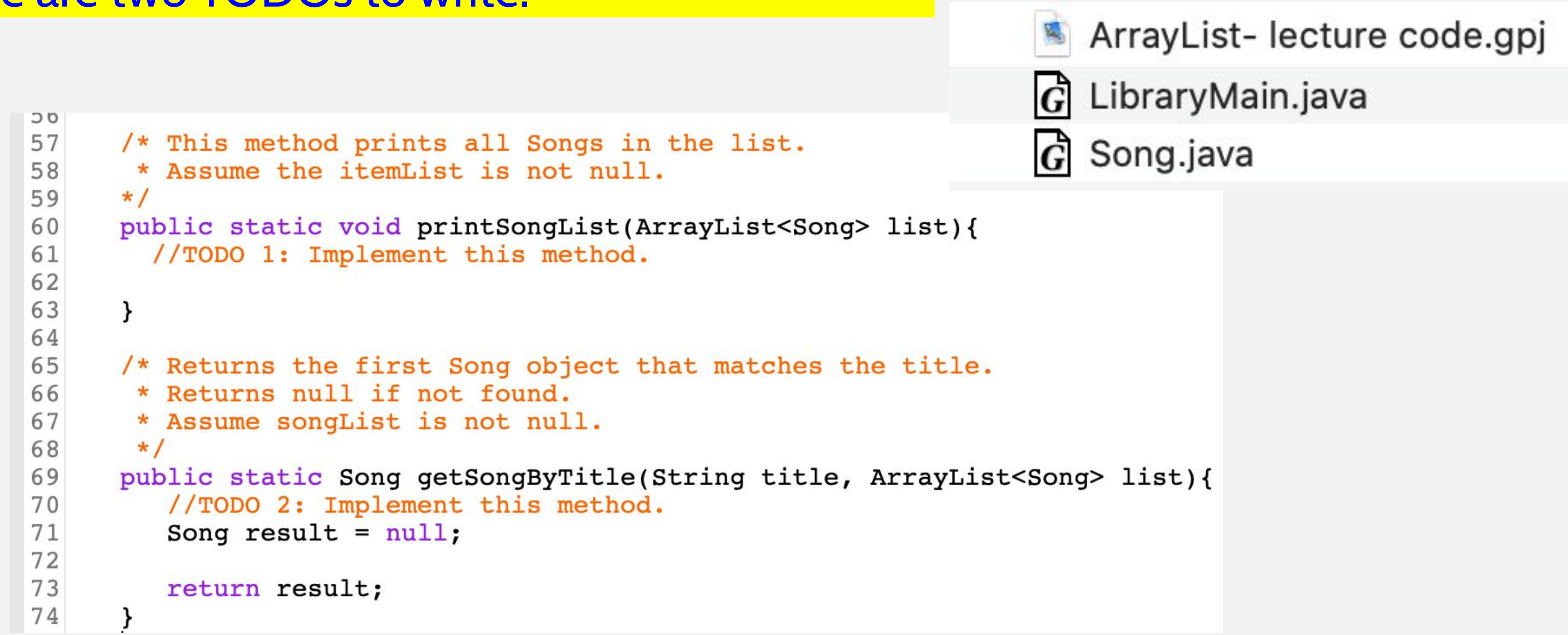
```
public static void removeElement(String targetStr, ArrayList<String> list){
    for(String curStr : list)
       if(targetStr.equals(curStr))
           list.remove(curStr);
}
```

```
ArrayList<String> strList = new ArrayList<String>();
strList.add("Red");
strList.add("Green");
strList.add("Blue");
System.out.println(strList);    // prints [Red, Green, Blue]
removeElement("Green", strList);
System.out.println(strList);   // prints [Red, Blue]
```

The list values were changed by the method.
That’s because the parameter is a ***reference*** to an Object.
This is true for ***any Object***, not just ArrayList.

ARRAY LIST DEMO IN JGrasp

An ArrayList is used to maintain a list of Songs.
There are two TODOs to write.



- ArrayList- lecture code.gpj
- LibraryMain.java
- Song.java

```java
    /* This method prints all Songs in the list.
     * Assume the itemList is not null.
    */
    public static void printSongList(ArrayList<Song> list){
      //TODO 1: Implement this method.

    }

    /* Returns the first Song object that matches the title.
     * Returns null if not found.
     * Assume songList is not null.
     */
    public static Song getSongByTitle(String title, ArrayList<Song> list){
       //TODO 2: Implement this method.
       Song result = null;

       return result;
    }
```

BYTE FROM COMPUTING HISTORY

See the fascinating timeline of Computer History.