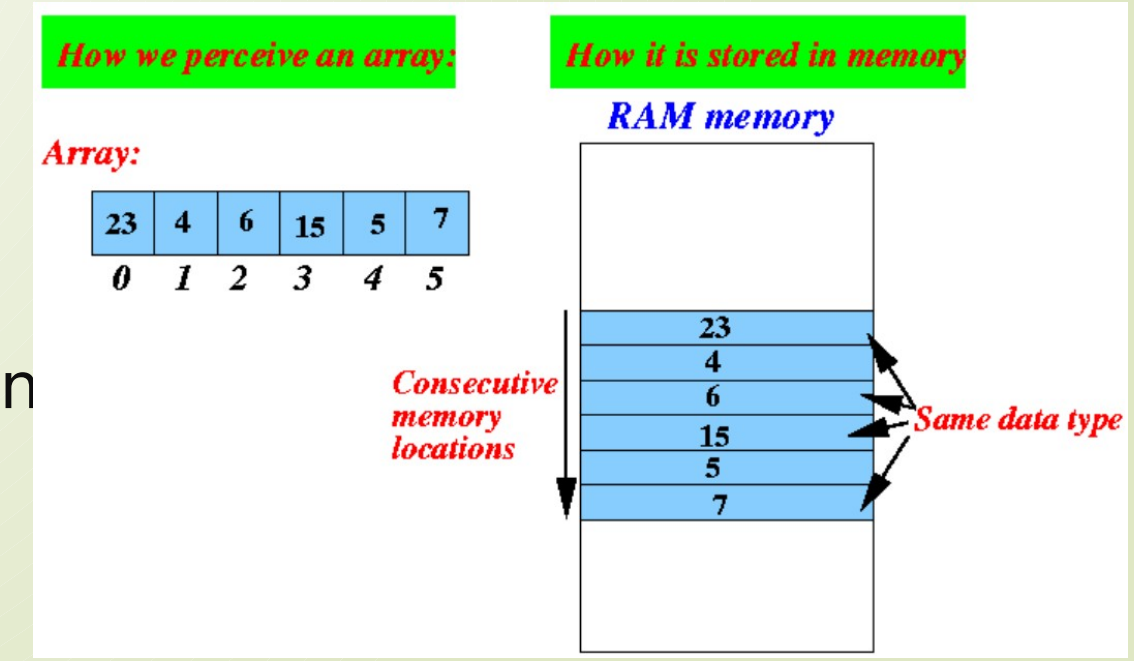
Array List and Wrapper Classes.

Topics

- ArrayLists
  - What is ArrayLists
  - Difference between Array and ArrayList
  - Advantages and disadvantages
  - Examples
- Object Wrapper Classes
  - What are wrapper classes
  - Why we need them
  - Concepts of Autoboxing/Unboxing.

ArrayList

- **Array + List**
- **Array:** An array in Java is a fixed-size collection of elements of the same type used for storing and manipulating data:
  - Fixed Size
  - Homogeneous Elements
  - Indexed Access
  - Contiguous Memory Allocation
  - Primitive Types and Objects



ArrayList (Cont.)

- **List:** In Java, a List is an interface that represents an ordered collection of elements.
- The List interface provides various methods for:
  - Adding elements
  - Removing elements
  - Accessing elements in the list.
  - Other book-keeping information
- A few common implementations of the List interface are:
  - ArrayList
  - LinkedList
  - Deque
  - Two other legacy classes implementing List are
    - Vector and Stack

What is an ArrayList in Java

- ArrayList is a data structure that allows dynamic resizing of array-based lists. Elements can be added or removed as needed.
- Some key features of ArrayList:
  - Resizable
  - Ordered Collection
  - Random Access
  - Allows Duplicates

Declaration of an ArrayList

An ArrayList is declared the following way:

```
ArrayList<Type> arrayListName = new ArrayList<>();
```

Or

```
ArrayList<Type> arrayListName = new ArrayList<Type>();
```

- Type: The type of elements that the ArrayList will contain.
- arrayListName: The name of the ArrayList variable.
- ArrayList<> or ArrayList<Type> : Indicates the ArrayList class from the java.util package.
  - The diamond operator (<>) is used for **type inference** and is used to specify the type of elements that the ArrayList will contain.
  - <Type> explicitly specifies the type parameter when creating the ArrayList using the constructor
- new ArrayList<>() or ArrayList<Type>(): Creates a new instance of the ArrayList.
  - The empty parentheses () indicate that the ArrayList is initially empty.

Declaration of ArrayList: Example

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Creating an ArrayList of strings
        ArrayList<String> arrayList = new ArrayList<>();

    // ArrayList<String> arrayList = new ArrayList<String>();
        // ArrayList<> arrayList = new ArrayList<String>();
        // ArrayList<> arrayList = new ArrayList<>();

    }
}
```

Type inference

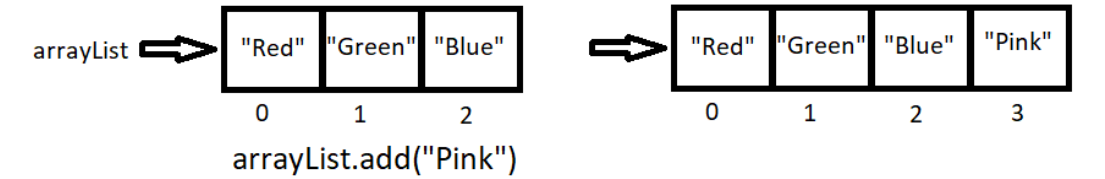
Ok – Explicit type

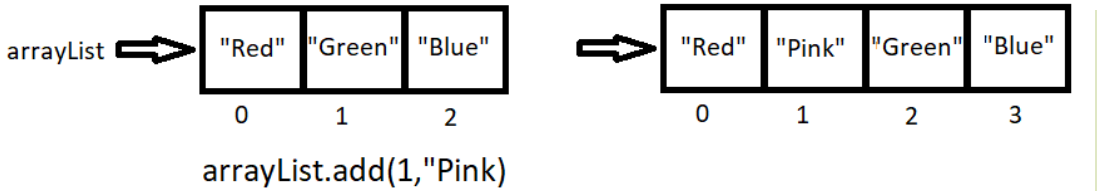
Compiler Error: illegal start of type

Compiler Error: illegal start of type

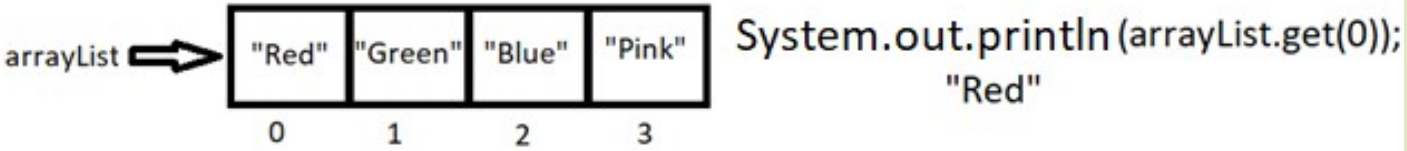
Common Operations on ArrayList:

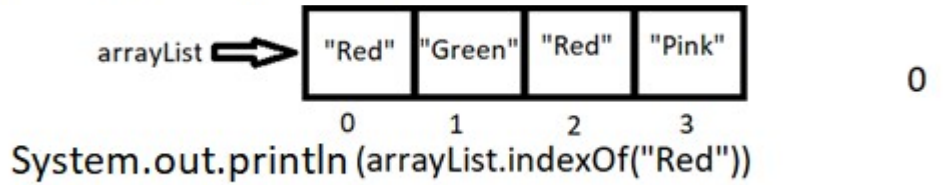
- **Adding Elements:**
  - **add**(E element): Adds the specified element to the end of the ArrayList.
  - **add**(int index, E element): Inserts the specified element at the specified position in the ArrayList, shifting the subsequent elements to the right.





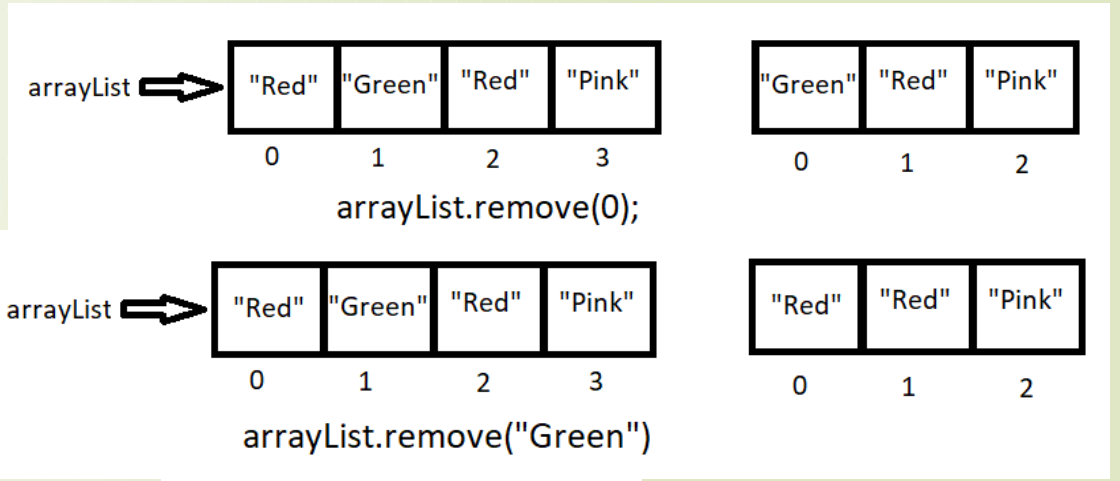
- **Accessing Elements:**
  - **get**(int index): Returns the element a specified index in the ArrayList.
  - **indexOf**(Object o): Returns the index of the first occurrence of the specified element in the ArrayList, or -1 if the element is not found.
  - **lastIndexOf**(Object o): Returns the index of the last occurrence of the specified element in the ArrayList, or -1 if the element is not found.

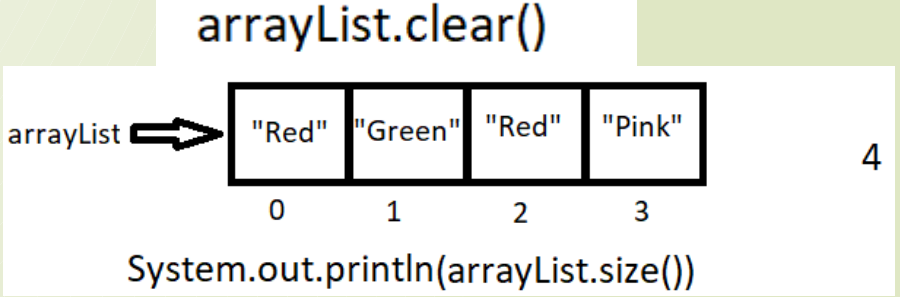




Common Operations on ArrayList (Cont.):

- **Removing Elements:**
  - **remove**(int index): Removes the element at the specified index from the ArrayList, shifting the subsequent elements to the left.
  - **remove**(Object o): Removes the first occurrence of the specified element from the ArrayList, if it is present.
  - **clear**(): Removes all elements from the ArrayList.
- **Checking Size and Empty Status:**
  - **size**(): Returns the number of elements in the ArrayList.
  - **isEmpty**(): Returns true if the ArrayList contains no elements, false otherwise.



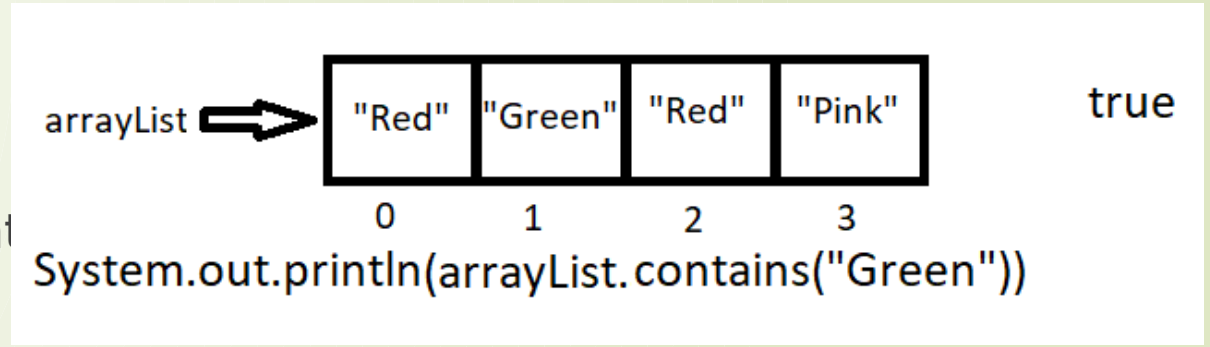


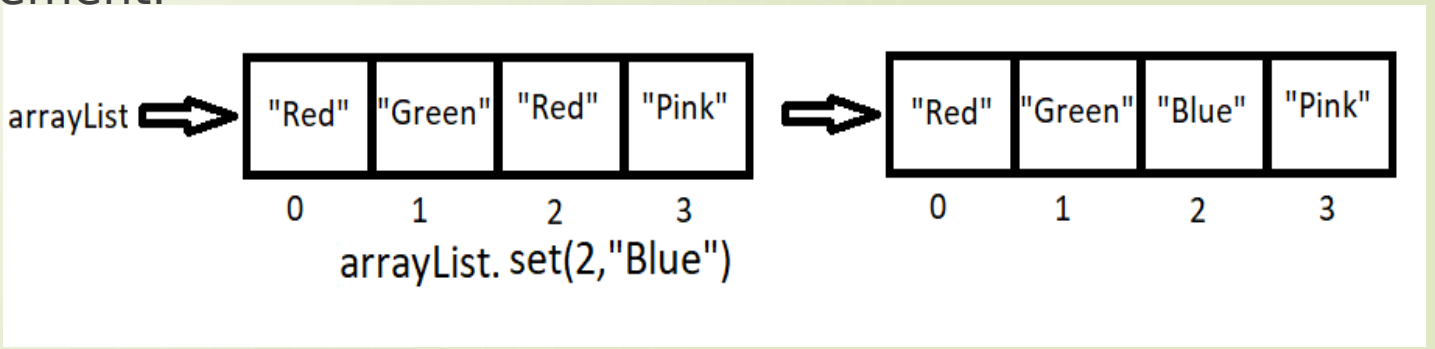

```
System.out.println(arrayList.isEmpty())     false

arrayList.clear();
System.out.println(arrayList.isEmpty())     true
```

Common Operations on ArrayList (Cont.):

- **Checking for Element Existence**:
  - **contains**(Object o): Returns true if the ArrayList contains the specified element false otherwise.
- **Replacing Elements:**
  - **set**(int index, E element): Replaces the element at the specified position in the ArrayList with the specified element.





Iterating over ArrayList elements

```
public static void main(String[] args) {
        // Create an ArrayList of colors
        ArrayList<String> colors = new ArrayList<>();

        //Adding colors
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        System.out.println("Colors in the ArrayList:");
        System.out.println("======================");

        System.out.println("Using enhanced for loop:");
        for (String color : colors) {
            System.out.println(color);
        }
        System.out.println();

        System.out.println("Using traditional for loop:");
        for (int i = 0; i < colors.size(); i++) {
            System.out.println(colors.get(i));
        }
```

```
Colors in the ArrayList:
======================
Using enhanced for loop:
Red
Green
Blue

Using traditional for loop:
Red
Green
Blue
```

Difference between Array and ArrayList in java

Some key differences between Arrays and ArrayLists :

- **Fixed Size vs. Dynamic Size:**
  - Arrays have a fixed size, while ArrayLists have a dynamic size
- **Primitive Types vs. Objects:**
  - Arrays can store both primitive types and objects, while ArrayLists can only store objects.
- **Access to Methods:**
  - Arrays do not have built-in methods for common operations like adding or removing elements while ArrayLists provide a rich set of methods for the same.
- **Performance:**
  - Arrays generally have better performance than ArrayLists. (More about it in the next slide)

Performance Comparison

Accessing Elements:

- Arrays generally provide faster access to elements because they use direct indexing.
- ArrayLists use an underlying array for storage, so accessing elements also has faster when using the get() method. However, accessing elements by index may involve additional overhead due to bounds checking and indexing calculations.

Insertion and Deletion:

- Arrays have fixed sizes, so inserting or deleting elements often requires shifting elements to accommodate the change, which can be time-consuming.
- ArrayLists dynamically resize themselves when elements are added or removed, which can lead to better performance for insertions and deletions compared to arrays.

Memory Overhead:

- Arrays have less memory overhead because they only store the elements themselves and do not require additional metadata.
- ArrayLists have more memory overhead because they use an underlying array along with additional metadata (such as the size of the list and the capacity of the underlying array).

Sorting an ArrayList

```
public static void main(String[] args) {
        // Create an ArrayList of colors
        ArrayList<String> colors = new ArrayList<>();

        //Adding colors
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        Collections.sort(colors);
        System.out.println("Sorted:");
        for (String item : colors) {
            System.out.println(item);
        }

    }
```

Sorted:
Blue
Green
Red

Convert an ArrayList into an Array

```
public static void main(String[] args) {
        // Create an ArrayList of colors
        ArrayList<String> colors = new ArrayList<>();

        //Adding colors
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        String[] array = colors.toArray(new String[colors.size()]);
        System.out.println("Array:");
        for (int i=0; i<array.length; i++) {
            System.out.println(array[i]);
        }

}
```

```
Array:
Red
Green
Blue
```

ArrayList containing Integers

```
public static void main(String[] args) {
        // Creating an ArrayList of integers
        ArrayList<int> numbers = new ArrayList<>();

        // Adding integers to the ArrayList
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        // Printing the ArrayList
        System.out.println("ArrayList of integers: " + numbers);
    }
```

```
Main.java:8: error: unexpected type
      ArrayList<int> numbers = new
ArrayList<>();
                       ^
  required: reference
  found:    int
1 error
```

```
public static void main(String[] args) {
        // Creating an ArrayList of integers
        ArrayList<Integer> numbers = new ArrayList<>();

        // Adding integers to the ArrayList
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        // Printing the ArrayList
        System.out.println("ArrayList of integers: " + numbers);
    }
```

```
ArrayList of integers: [10, 20, 30]
```

Wrapper Class

- In Java, wrapper classes are the classes that wrap or encapsulate primitive data types to be treated as an object.
  - They provide a way to convert primitive data types into objects.
  - Thus, allow them to be used where objects are required, for example,
    - ArrayList or LinkedList
    - Generics,
    - Methods that require objects as arguments.
- Each wrapper class provides methods and constructors to convert between primitive data types and objects of the wrapper class.
  - For example:
    - We can use the valueOf() method to create a wrapper object from a primitive value
    - We can use methods like intValue(), doubleValue(), etc., to extract the primitive value from the wrapper object.

Example of Wrapper Class

```
public static void main(String[] args) {
        // Create an Integer object from a primitive int
        Integer num1 = Integer.valueOf(10);

        // Create a Double object from a primitive double
        Double num2 = Double.valueOf(3.14);

        // Convert Integer object to int
        int intValue = num1.intValue();
        System.out.println("intValue: " + intValue);

        // Convert Double object to double
        double doubleValue = num2.doubleValue();
        System.out.println("doubleValue: " + doubleValue);
    }
```

```
intValue: 10
doubleValue: 3.14
```

List of Wrapper Classes in Java

Primitive type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

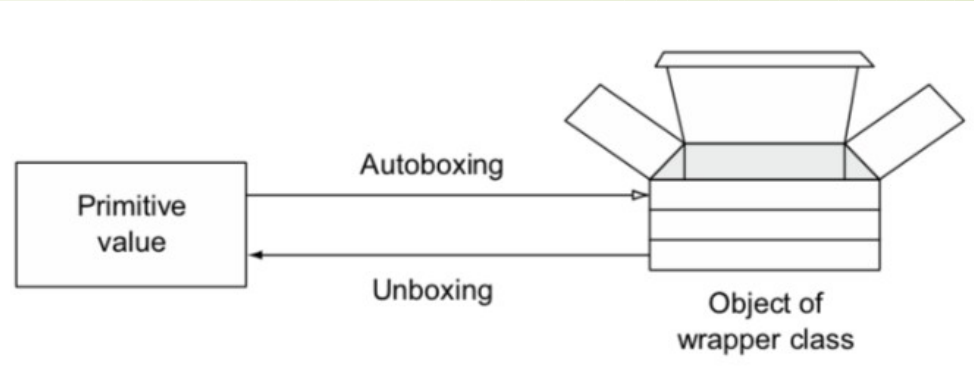
Explicit and Implicit Conversion

```
// Create an Integer object from a primitive int
Integer num1 = Integer.valueOf(10);
// Create a Double object from a primitive double
Double num2 = Double.valueOf(3.14);
// Convert Integer object to int
int intValue = num1.intValue();
// Convert Double object to double
double doubleValue = num2.doubleValue();
```

- When we call some special methods to convert a primitive value into its wrapper class object or a wrapper class object into its primitive value, it is called **explicit conversion.**
- When these conversions take place automatically (without any special method), it is called implicit conversion. It is also known as **autoboxing and unboxing.**

Autoboxing and Unboxing

- Autoboxing and unboxing automatically convert between primitive data types and their wrapper classes.
  - It simplifies the process by reducing the need for manual conversion.
    - We do not need to use the valueOf() method to create a wrapper object from a primitive value
    - We do not need to use methods like intValue(), doubleValue(), etc., to extract the primitive value from the wrapper object.



Autoboxing and Unboxing: Example

Autoboxing:

- When we assign an int value to an Integer variable, autoboxing automatically converts the int value to an Integer object.

```
// Autoboxing: converting int to Integer
int intValue = 10;
Integer integerValue = intValue; // Autoboxing
```

Unboxing:

- when we assign an Integer object to an int variable, unboxing automatically extracts the int value from the Integer object.

```
// Unboxing: converting Integer to int
Integer integerValue = new Integer(20);
int intValue = integerValue; // Unboxing
```

A Few Practice Problem

- **Sum of Elements:** Write a program to find the sum of all elements in an ArrayList of integers.
- **Remove Duplicates:** Write a program to remove duplicates from an ArrayList of strings.
- **Find Maximum and Minimum:** Write a program to find the maximum and minimum elements in an ArrayList of integers.
- **Search Element:** Write a program to search for an element in an ArrayList and return its index.
- **Split Even and Odd Numbers:** Write a program to split an ArrayList of integers into two separate lists: one containing even numbers and the other containing odd numbers.
- **Merge Two Sorted Lists:** Write a program to merge two sorted ArrayLists into a single sorted ArrayList.