COSC 121

Computer Programming II

ArrayLists and Intro to Generics

Part 2/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Outline

Previously:

- Intro to Java Collection Framework
- The `ArrayList` Class
- Implementing a `Stack` using `ArrayList`
- Sample applications

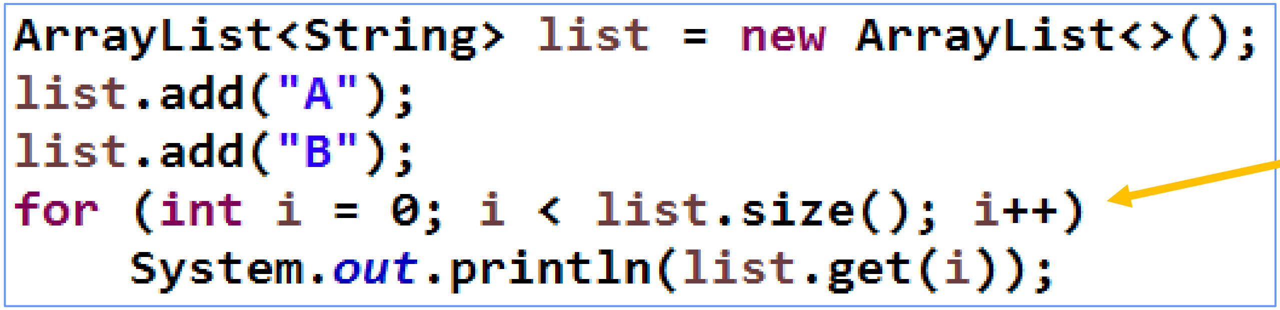
Today:

- Iterating Over an `ArrayList`
- Random Access in `ArrayLists`
- Useful Methods for Lists
  - `Arrays` and `Collections` classes
- Intro to Generics

Iterating Over an `ArrayList`

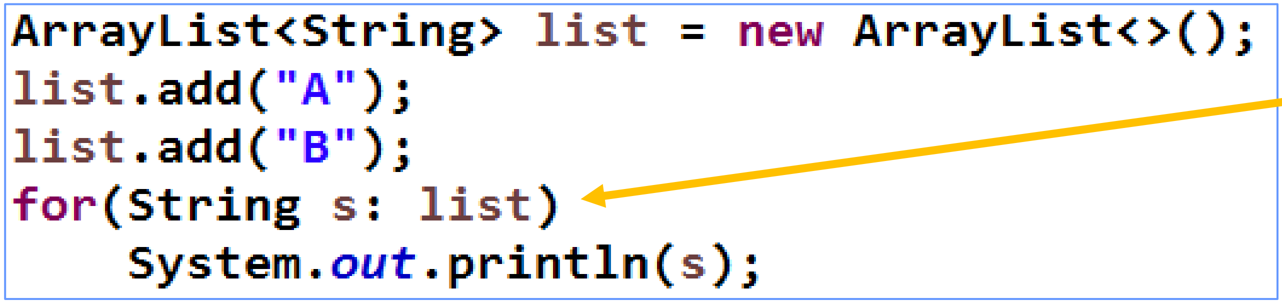
Iterating Over an `ArrayList`

You can iterate over this array in different ways:



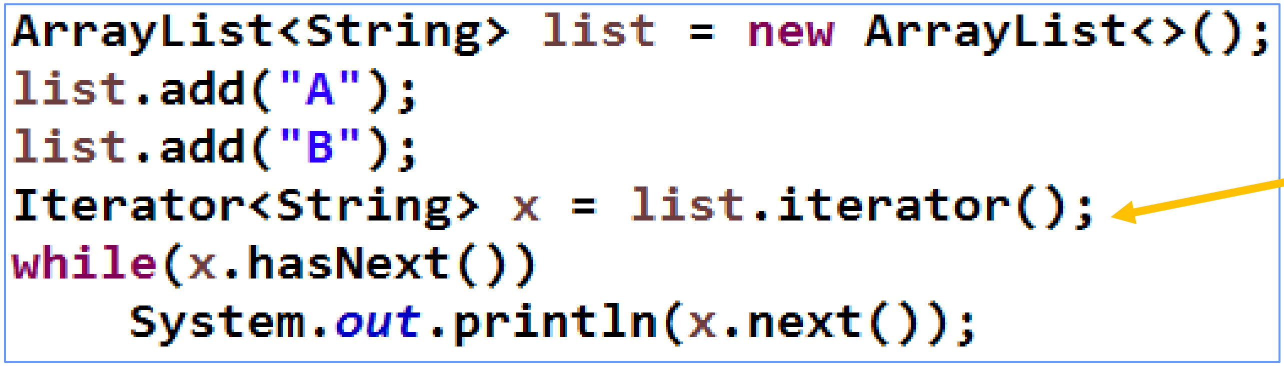
```java
ArrayList<String> list = new ArrayList<>();
list.add("A");
list.add("B");
for (int i = 0; i < list.size(); i++)
    System.out.println(list.get(i));
```

Using the index



```java
ArrayList<String> list = new ArrayList<>();
list.add("A");
list.add("B");
for(String s: list)
    System.out.println(s);
```

Using for-each



```java
ArrayList<String> list = new ArrayList<>();
list.add("A");
list.add("B");
Iterator<String> x = list.iterator();
while(x.hasNext())
    System.out.println(x.next());
```

Using an iterator.

Iterators and `ArrayList`

An `ArrayList` (and each collection) is **`Iterable`**. You can obtain its ***`Iterator`*** to traverse all its elements.

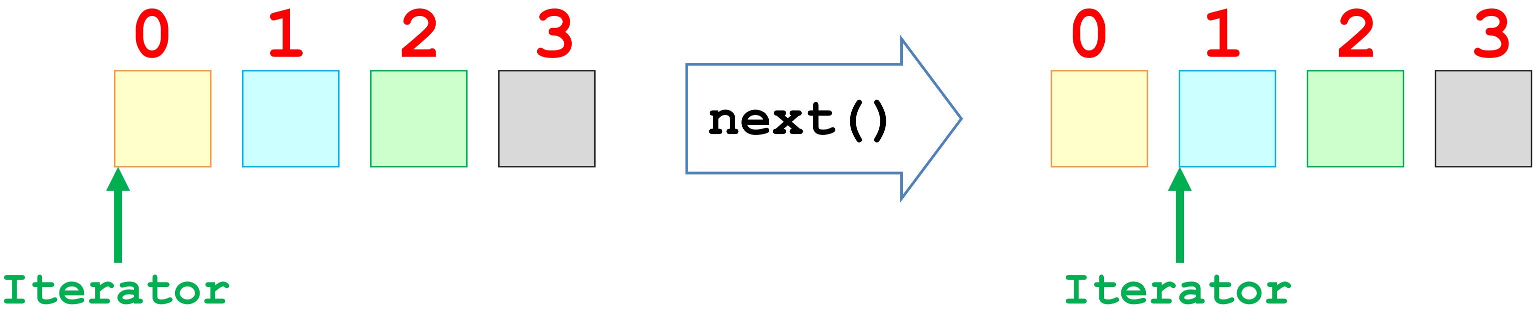
Two more useful methods in `ArrayList`:

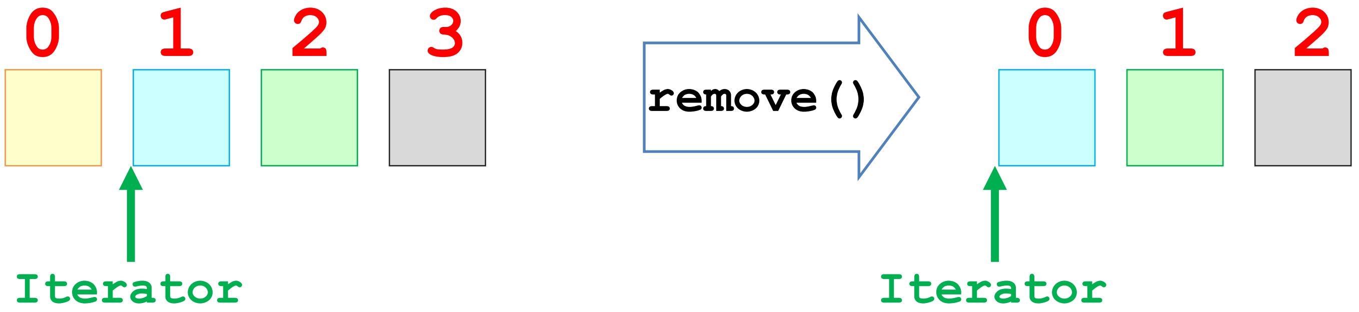
- **`iterator()`**
- **`listIterator()`**

Iterators and ArrayList, cont.

`iterator()`: returns an iterator object

- The iterator object implements the Iterator interface.
- Methods:
  - `hasNext()`: Returns true if this iterator has more elements to traverse.
  - `next()`: Returns the next element from this iterator.
  - `remove()`: Removes the last element obtained using the next method.

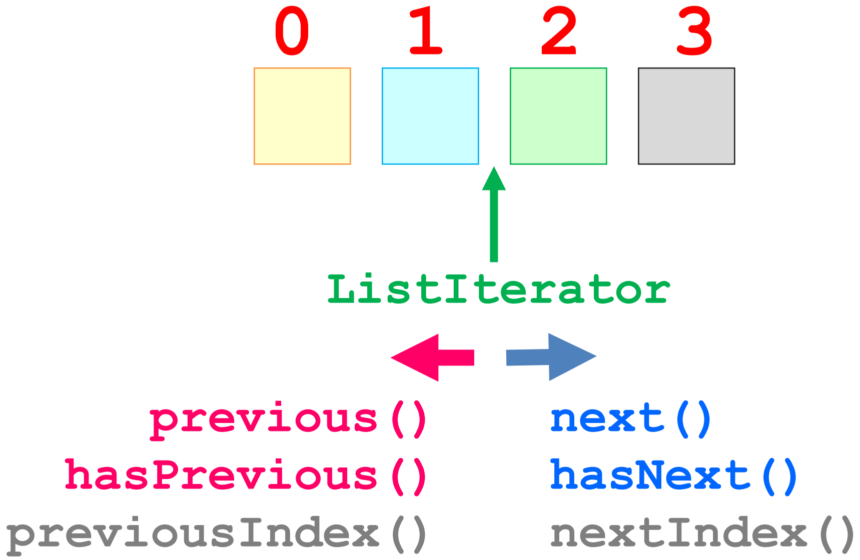




Iterators and ArrayList, cont.

listIterator(): returns a ListIterator object

- ListIterator is a subtype of Iterator
- Methods:
  - **next()**, **previous()**
  - **hasNext()**, **hasPrevious()**
  - **remove()**
  - **nextIndex()**, **previousIndex()**   returns the next/previous index

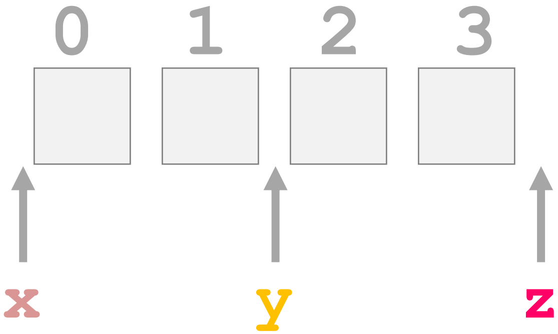


Iterators and `ArrayList`, cont.

`listIterator()` also allows initializing the iterator with an index

- To position the iterator some place other than the beginning of the list.

```
ListIterator x = list.listIterator(0); //same as no arg
ListIterator y = list.listIterator(2);
ListIterator z = list.listIterator(list.size());
```

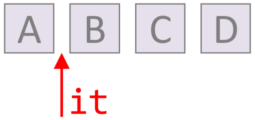


Iterators and `ArrayList`, cont.

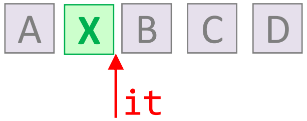
`ListIterator.add()`: inserts just before the iterator.

`ListIterator.set()`: replaces last element returned by next or previous

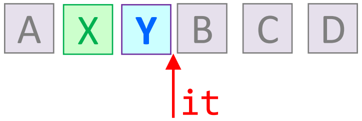
```
ListIterator<String> it;
ArrayList<String> list = new ArrayList<>(Arrays.asList("A","B","C","D"));
it = list.listIterator(1);
```



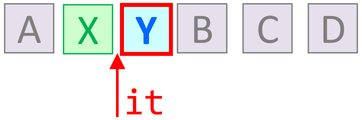
```
it.add("X");
```



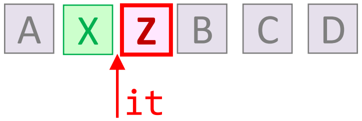
```
it.add("Y");
```



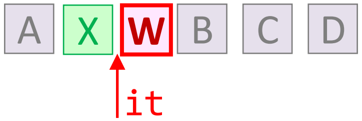
```
it.previous();
```



```
it.set("Z");
```



```
It.set("W");
```



Practice 4

Find the sum of all elements in array list, `nums`, using three different ways (loops).

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1,2,3,4));

//using for-i loop
int sum = 0;
for (int i = 0; i < list.size(); i++)
  sum += list.get(i);

//using for-each loop
sum = 0;
for (int item : list)
  sum += item;

//using iterator
sum = 0;
Iterator<Integer> it = list.iterator();
while(it.hasNext())
  sum += it.next();
```

Practice 5

Use a ListIterator to print all elements in an ArrayList in forward direction **then** in backward direction.

- e.g., for [3, 2, 6, 9], the output would be:

  Elements in forward direction: 3 2 6 9

  Elements in backward direction: 9 6 2 3

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(3,2,6,9));

ListIterator<Integer> it = list.listIterator();

//forward printing
while(it.hasNext())
  System.out.print(it.next() + " ");

System.out.println();

//backward printing
while(it.hasPrevious())
  System.out.print(it.previous() + " ");
```

Practice 6

Use a `ListIterator` to print all elements in an `ArrayList` in backward direction.

- e.g., for [3, 2, 6, 9], the output would be 9 6 2 3

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(3,2,6,9));

ListIterator<Integer> it = list.listIterator(list.size());

//backward printing
while(it.hasPrevious())
  System.out.print(it.previous() + " ");
```

Removing all A’s

Suppose `ArrayList x` has 4 strings `["A","A","B","A"]`.

Write code the will remove all A’s.

Solution 1:

Efficient?

```
while(x.contains("A"))
        x.remove("A");
```

Solution 2:

Efficient?

```
Iterator <String> it = x.iterator();
while(it.hasNext())
        if(it.next().equals("a"))
                it.remove();
```

Solution 3:

Argument must be a Collection

```
x.removeAll( Arrays.asList("A") );
```

Experiment

```
final int N = 20000; //try other values
//create an arraylist initialized to random chars then "a"s
ArrayList<String> list1 =  new ArrayList<>(N);
for(int i = 0; i<N; i++)
   if(i<N/2) list1.add((char)(Math.random()*25+'b')+""); //random characters
   else      list1.add("a");
//create a second arraylist identical to list1
ArrayList<String> list2=(ArrayList<String>)list1.clone();

//***Remove all "a"s and measure the time ***

//method 1: using contains()
long start = System.currentTimeMillis();
while(list1.contains("a"))   //search for a
   list1.remove("a");        //search for a, then remove a
long end = System.currentTimeMillis();
System.out.printf("Method 1 Time: %d ms\n",(end-start));

//method 2: using an iterator
start = System.currentTimeMillis();
Iterator<String> it = list2.iterator();
while(it.hasNext())
   if(it.next().equals("a"))
      it.remove();  //remove a
end = System.currentTimeMillis();
System.out.printf("Method 2 Time: %d ms\n", (end-start));
```

Output

```
Method 1 Time: 1269 ms
Method 2 Time: 9 ms
```

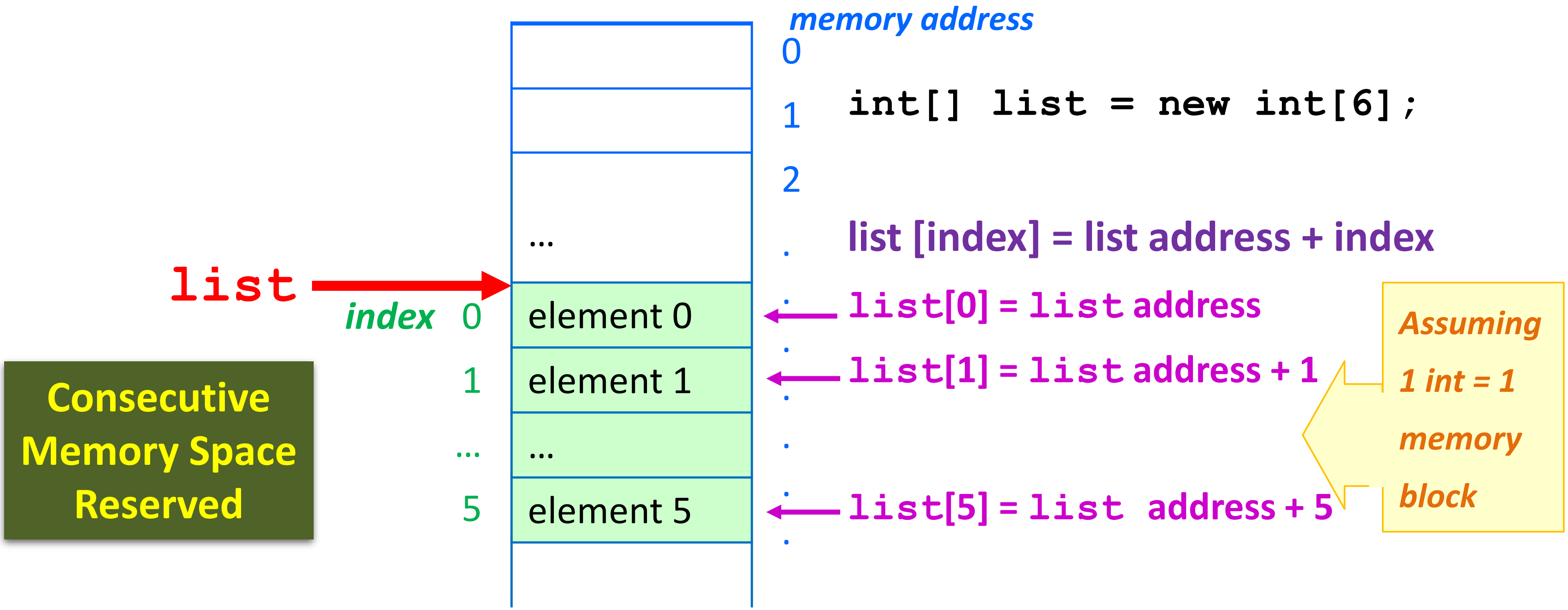
Random Access in `ArrayList`

`ArrayList` Supports Random Access

Arraylists are **implemented using arrays**.

- Whenever the current array cannot hold new elements in the list, a ***larger new array is created*** to replace the current array.
  - Illustration: http://cs.armstrong.edu/liang/animation/web/ArrayList.html

Arrays use an ***index*** to reference its elements



```
int[] list = new int[6];
```

list [index] = list address + index

`list`[0] = `list` address

`list`[1] = `list` address + 1

`list`[5] = `list` address + 5

Assuming 1 int = 1 memory block

Experiment

```
//create and initialize an arraylist
ArrayList<String> list =  new ArrayList<>(20000);
for(int i = 0; i<list.size(); i++)
    if(i<N/2)  list.add((char)(Math.random()*25+'b')+"");
    else       list.add("a");

//***Visiting all elements***
//method 1: using get(index), O(n)
long start = System.currentTimeMillis();
for(int i = 0; i<list.size(); i++)
    list.get(i);   // random access – very efficient!
long end = System.currentTimeMillis();
System.out.printf("Method 1 Time: %d ms\n",(end-start));

//method 2: using an iterator, O(n)
start = System.currentTimeMillis();
Iterator<String> it = list.iterator();
while(it.hasNext())
    it.next();
end = System.currentTimeMillis();
System.out.printf("Method 2 Time: %d ms\n",(end-start));
```

Output

```
Method 1 Time: 2 ms
Method 2 Time: 2 ms
```

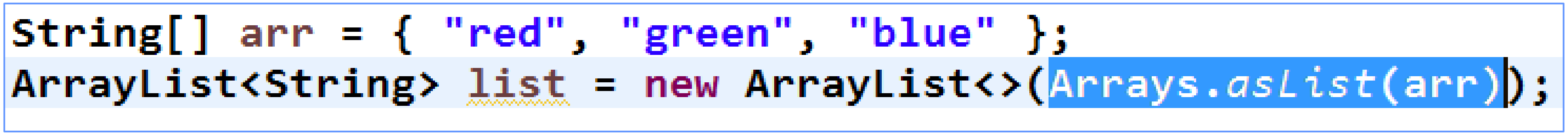
Useful Methods for Lists

(`Arrays` and `Collections` classes)

Array ↔ ArrayList

Array → ArrayList

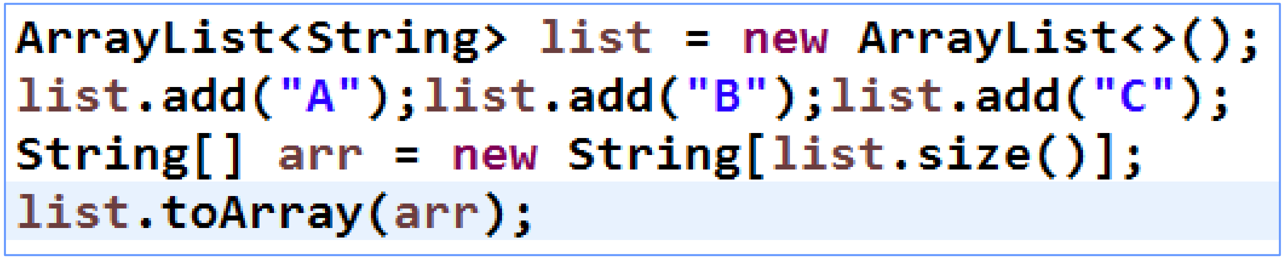
- Creating an `ArrayList` from an array:
- Syntax: `list = Arrays.asList(array)`



```
String[] arr = { "red", "green", "blue" };
ArrayList<String> list = new ArrayList<>(Arrays.asList(arr));
```

ArrayList → array

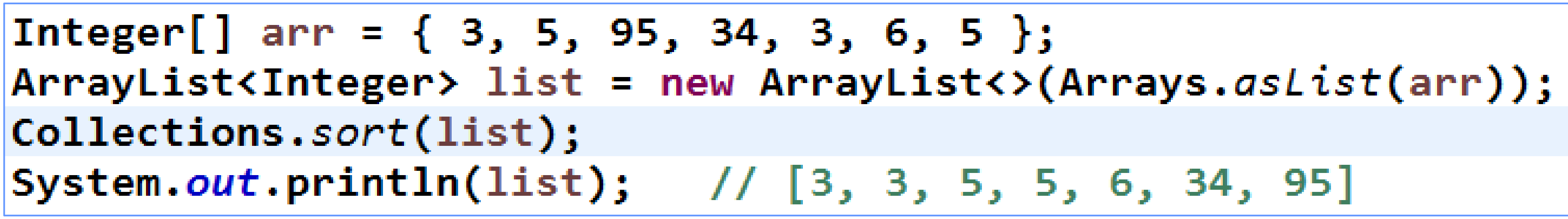
- Creating an array from an `ArrayList`:
- Syntax: `list.toArray(array);`



```
ArrayList<String> list = new ArrayList<>();
list.add("A");list.add("B");list.add("C");
String[] arr = new String[list.size()];
list.toArray(arr);
```

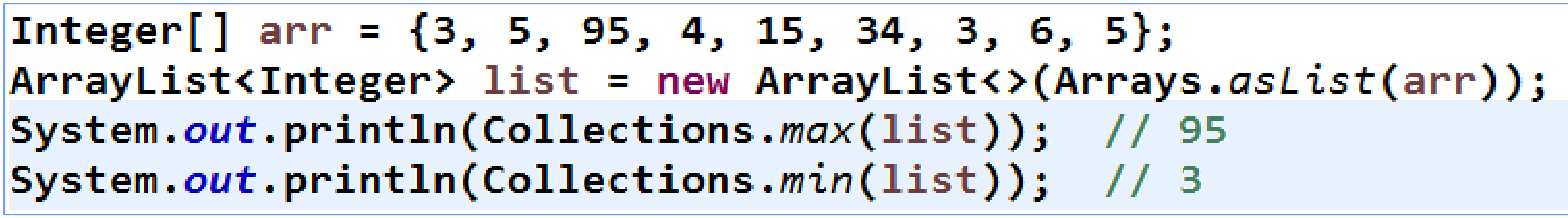
java.util.Collections methods

Collections.sort (if elements are **comparable**):



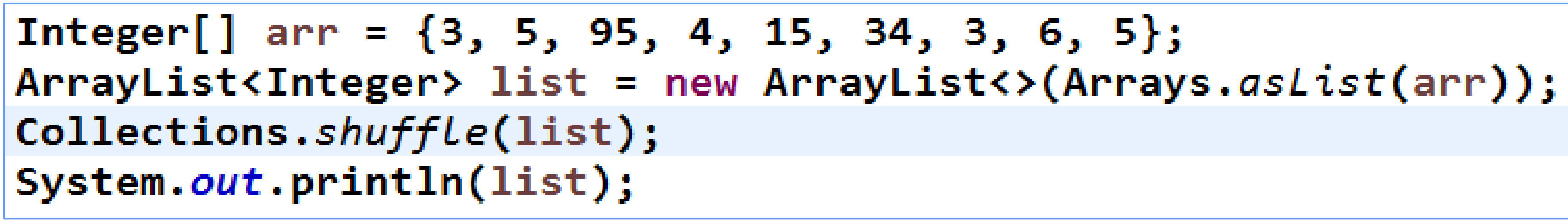
```
Integer[] arr = { 3, 5, 95, 34, 3, 6, 5 };
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(arr));
Collections.sort(list);
System.out.println(list);    // [3, 3, 5, 5, 6, 34, 95]
```

Collections.min or .max



```
Integer[] arr = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(arr));
System.out.println(Collections.max(list));  // 95
System.out.println(Collections.min(list));  // 3
```

Collections.shuffle



```
Integer[] arr = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(arr));
Collections.shuffle(list);
System.out.println(list);
```

Intro to Generics

Generics

A generic class has at least one member of an *unspecified type*.

`ArrayList` is a **generic class** with a generic type **`E`**

- We have also seen generics before, e.g., the Comparable interface.
  `class Robot implements Comparable<Robot>`
- The type(s) you provide on instantiation appear in the API as single letters in angle brackets after the name of the class, e.g. `ArrayList<E>`.

All collections support generic (or parameterized) types to indicate what type is stored in the collection.

It is better to precisely **specify the type of objects** in a collection so that the compiler can check for errors.

- If you don't, then a **collection can store any type of object** as all objects are a subclass of **`Object`**.

Object Wrapper Classes

The generic type **E** **must be reference type**.

- You cannot replace a generic type with a primitive type such as **int**, **double**, or **char**.

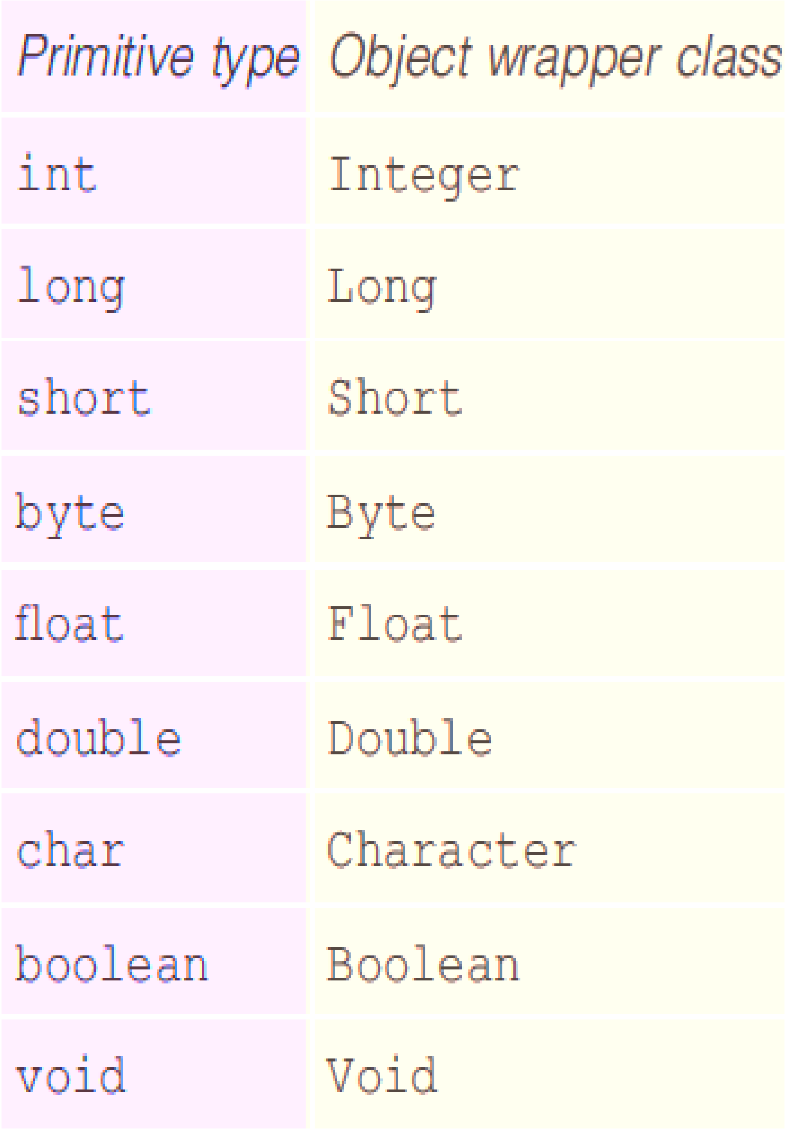
For example, the following statement is wrong:

~~ArrayList<**int**> a = **new** ArrayList<>();~~ ***// X***

To create an ArrayList object for int values, you have to use:

ArrayList<Integer> a =new ArrayList<>(); //✓

Java employed ***object wrappers***, which 'wrap around' or encapsulate all the primitive data types and allow them to be treated as objects.



Primitive type	*Object wrapper class*
int	Integer
long	Long
short	Short
byte	Byte
float	Float
double	Double
char	Character
boolean	Boolean
void	Void

But, how am I still able to insert primitives into a list?

If you have

```
ArrayList<Double> a = new ArrayList<>();
```

You can add an **double** value to **a**. For example,

```
a.add(5.5);
```

Java automatically **wraps 5.5 into new Double(5.5)**. This is called ***auto-boxing***

Examples:

```
list.add(5.5); // 5.5 is automatically converted to new Double(5.5)

list.add(3.0); // 3.0 is automatically converted to new Double(3.0)

Double doubleObj = list.get(0); // No casting, returns Double

double d = list.get(1); // Automatically converted to double
```

Index or Value-of-Item??

When using generics, **exact PARAMETER MATCHING** takes precedence over **AUTO-BOXING**

Consider this code below:

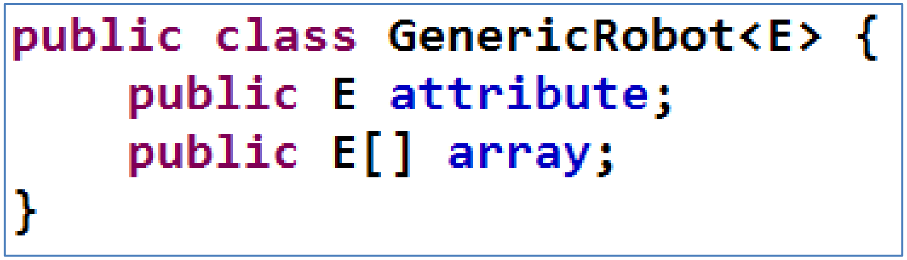
```
ArrayList<Integer> list = new ArrayList<>();
list.add(3);         //ok, auto-boxing used for 3
list.add(0,9);       //ok, auto-boxing used for 9
list.remove(9);      //ERROR. 9 is not valid index
```

- Why 9 is considered an index but not a value?
  - That is because exact parameter matching tells Java that `9` as an `int` matches remove(int index) - but not remove(Object value), and therefore it doesn’t try to ‘auto-box’ 9 in an Integer class and simply assumes 9 is an index.

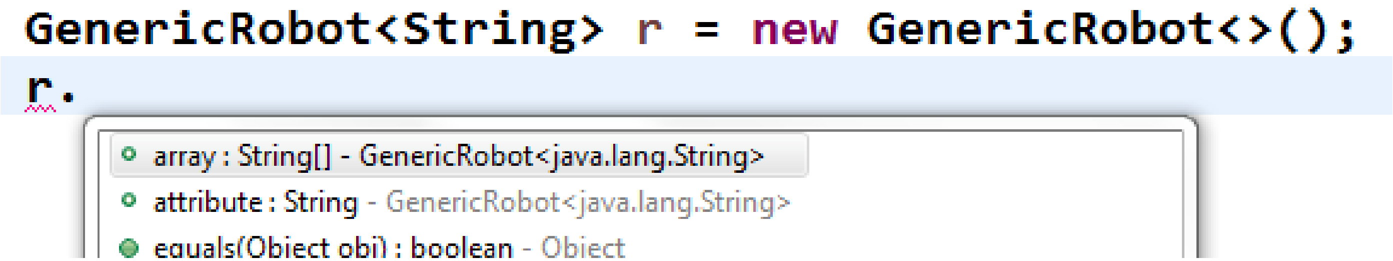
Defining Generic Classes and Interfaces

Define a generic class Generic Robot that has two attributes:

- One instance variable of the generic type
- One instance variable of the generic **array** type

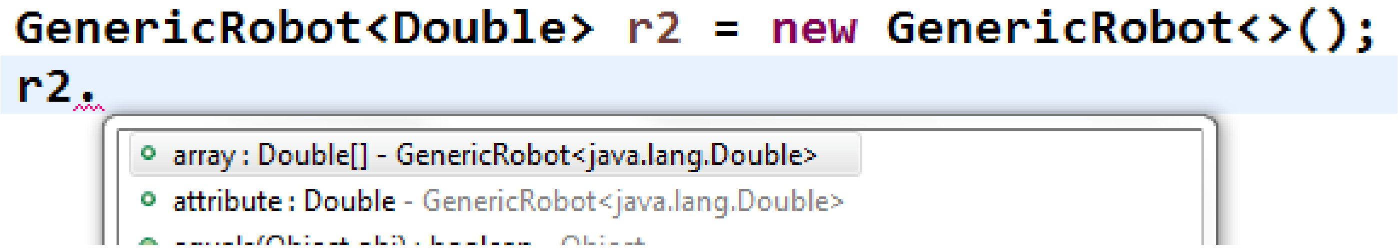


```
public class GenericRobot<E> {
    public E attribute;
    public E[] array;
}
```



```
GenericRobot<String> r = new GenericRobot<>();
r.
```

- array : String[] - GenericRobot<java.lang.String>
- attribute : String - GenericRobot<java.lang.String>
- equals(Object obj) : boolean - Object



```
GenericRobot<Double> r2 = new GenericRobot<>();
r2.
```

- array : Double[] - GenericRobot<java.lang.Double>
- attribute : Double - GenericRobot<java.lang.Double>
- equals(Object obj) : boolean - Object

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc.

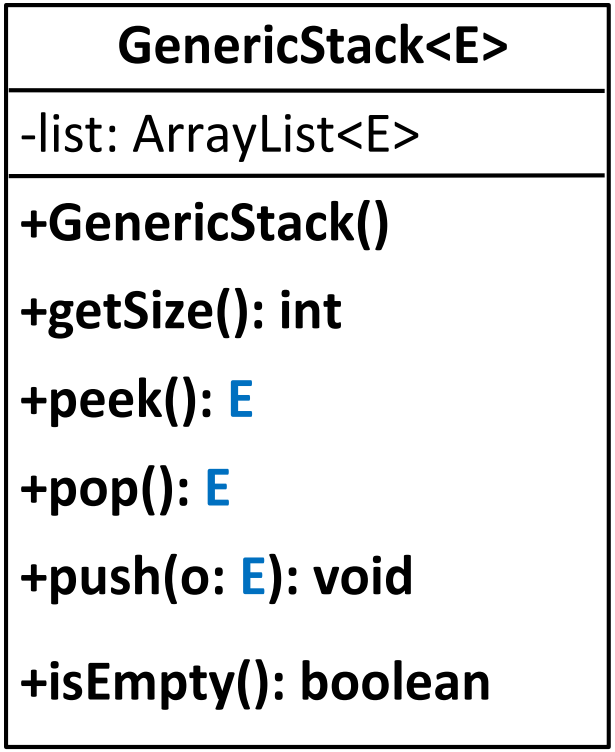
Defining Generic Classes and Interfaces

You have seen before how to create `MyStack` class that holds instances of the type `Object`, which means any type (you can store *cars*, *apples*, and *humans* in the same stack).

To force `MyStack` to accept only a certain type of objects, you **two options**:

- 1) to create individual classes for each type (e.g., *Human*Stack, *Apple*Stack, *Car*Stack, etc) – bad approach!!
- 2) to **create a generic class**, GenericStack<E>, where E is replaced by the required type when creating an instance of the class.

Defining Generic Classes and Interfaces



GenericStack<E>

-list: ArrayList<E>

+GenericStack()

+getSize(): int

+peek(): E

+pop(): E

+push(o: E): void

+isEmpty(): boolean

A list to store elements.

Generates an empty stack

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns true if this stack is empty.

Defining Generic Classes and Interfaces

```
public class MyStack<E> {
    private ArrayList<E> list = new ArrayList<>();

    public int size() {return list.size();}
    public boolean isEmplty() {return list.isEmpty();}

    public void push(E e) {list.add(e);}
    public E pop() {
        return size()>0? list.remove(list.size()-1):null;
    }
    public E peek(){
        return size()>0? list.get(list.size()-1):null;
    }

    public static void main(String[] args) {
        MyStack<String> stack = new MyStack<>();
        stack.push("A"); stack.push("B");
        System.out.println(stack.peek());
    }
}
```

What to know more about Generics?

There is a lot more to discuss about Java Generics. However, they are outside the scope of this course.

More can be found in

- Chapter 19 of the textbook
- COSC 222: Data Structures