Topic 8: Data Structures

Programming Practice and Applications (4CCS1PPA)

Dr. Asad Ali
Thursday 24th November
programming@kcl.ac.uk

DATA STRUCTURES: OBJECTIVE

To understand **more complex** data structures that abstract away the details, and provide various ways, of organising collections of elements.

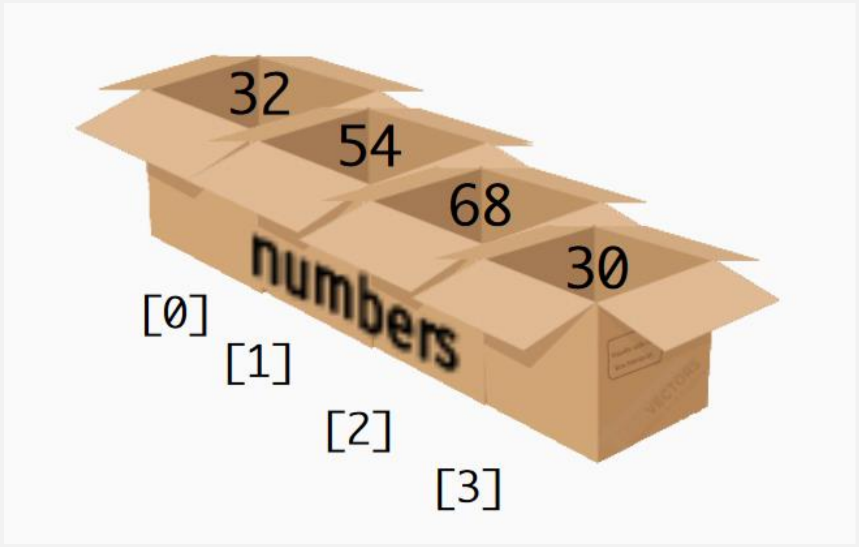
ARRAYS: GREAT BUT LIMITED

Arrays enable the storage of multiple values (all of the same type) in a single variable.

Instead of a single space for a value, a box can have multiple compartments.

There are a **fixed number** of compartments.

The compartments are labelled with an **index** number, in ascending order **starting at 0**.



ARRAYS: GREAT BUT LIMITED

The key problem with arrays is that they have a **fixed** length.

What happens if you want to add more values than the length?

ARRAYS: GREAT BUT LIMITED

The key problem with arrays is that they have a **fixed** length.

What happens if you want to add more values than the length?

- Create a new array bigger than the old one, and copy across the content

What happens if you remove an element from the array?

ARRAYS: GREAT BUT LIMITED

The key problem with arrays is that they have a **fixed** length.

What happens if you want to add more values than the length?

- Create a new array bigger than the old one, and copy across the content

What happens if you remove an element from the array?

- Array is partially-filled, so keep track of the last non-empty index

In summary: Headaches

Good news: You do not have to use arrays all the time!

DATA STRUCTURES

The Java libraries provides various useful **data structures** for storing a number of elements.

These abstract away the management of the details of the collection, such as its size and the order of the elements.

These data structures are classes that you must **import** into your program to use.

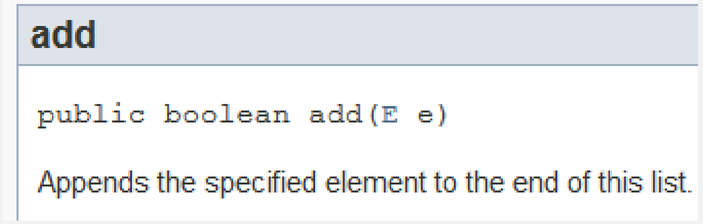
The library where these data structures are contained is **java.util** and to import the class you wish to use, you must append to this a '**.**' followed by the name of the class.

ARRAYLISTS

One of the classes in this library is the **class ArrayList** which works like an array, but **does not** have a fixed length.

Crucially, it **hides** the complications of adding and removing elements from the ArrayList.

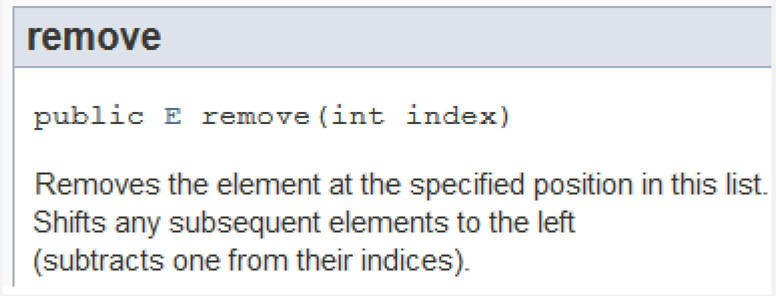
So, it provides a simple interface for these operations (and more):



add

```
public boolean add(E e)
```

Appends the specified element to the end of this list.



remove

```
public E remove(int index)
```

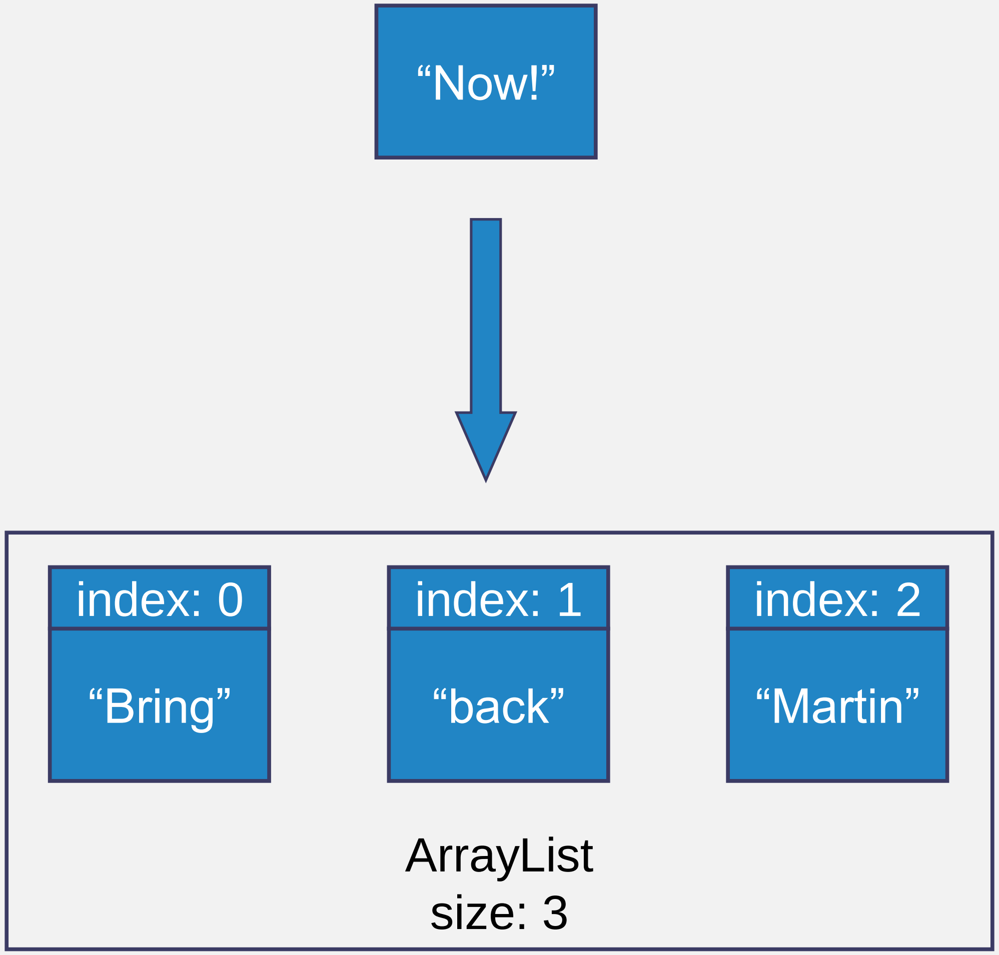
Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices).

Remember, it must be imported into your program:

```
import java.util.ArrayList
```

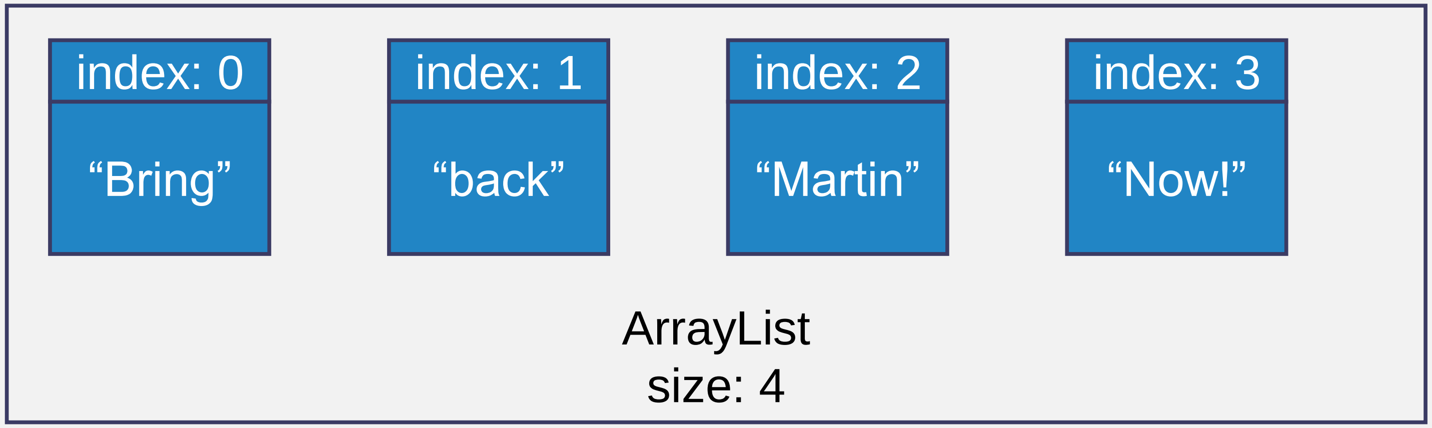
ARRAYLIST: ADD

Add an element (to the end of the ArrayList):



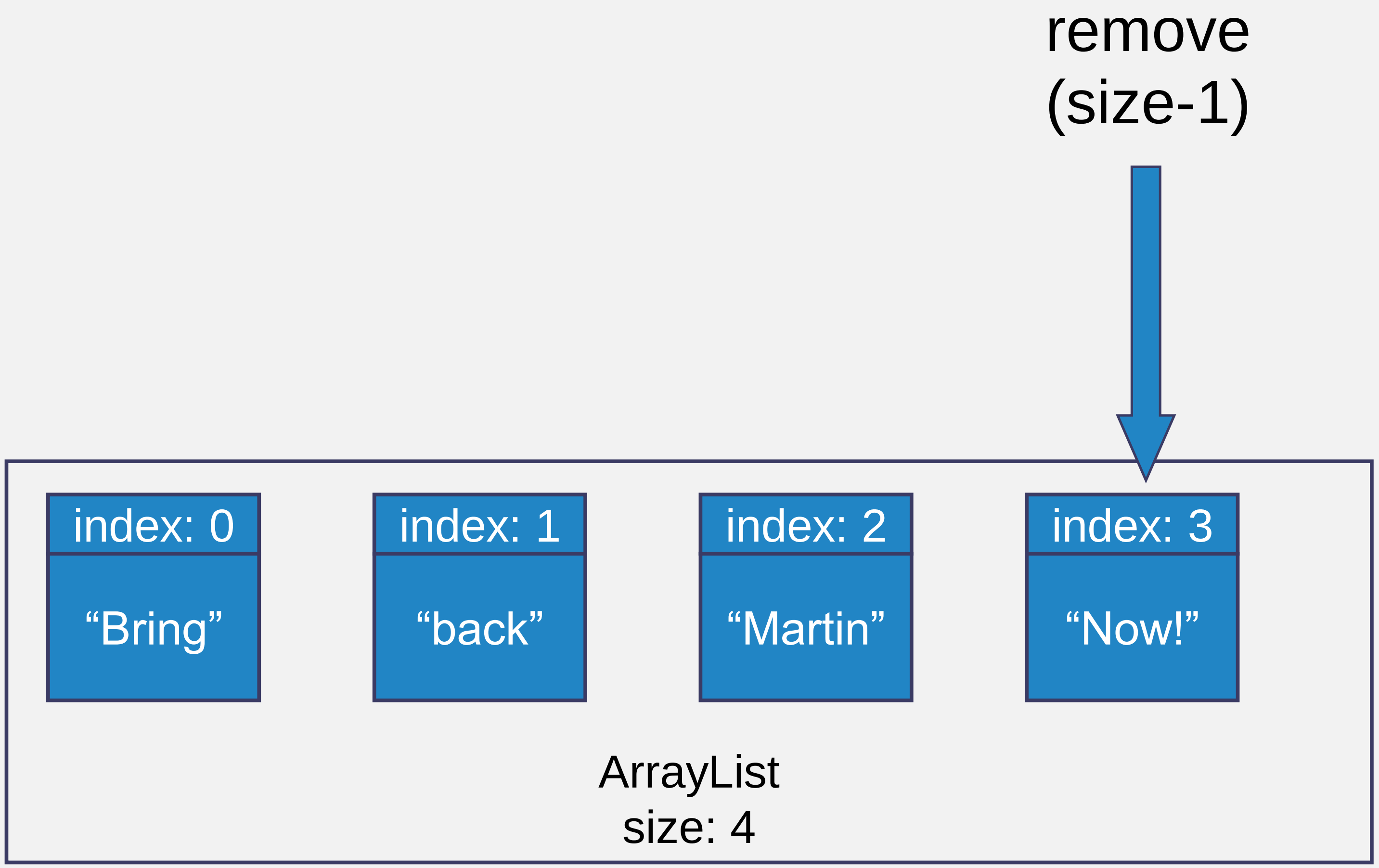
ARRAYLIST: ADD

Add an element (to the end of the ArrayList):



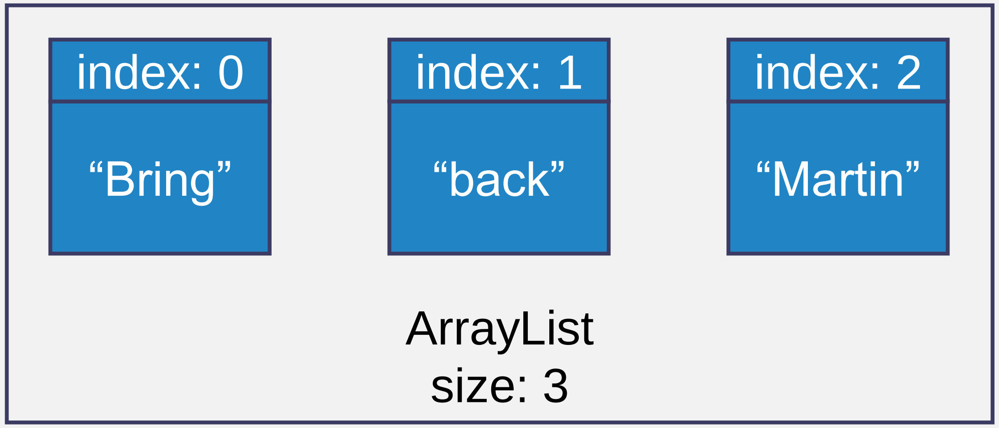
ARRAYLIST: REMOVE

Remove an element from the end of the ArrayList:



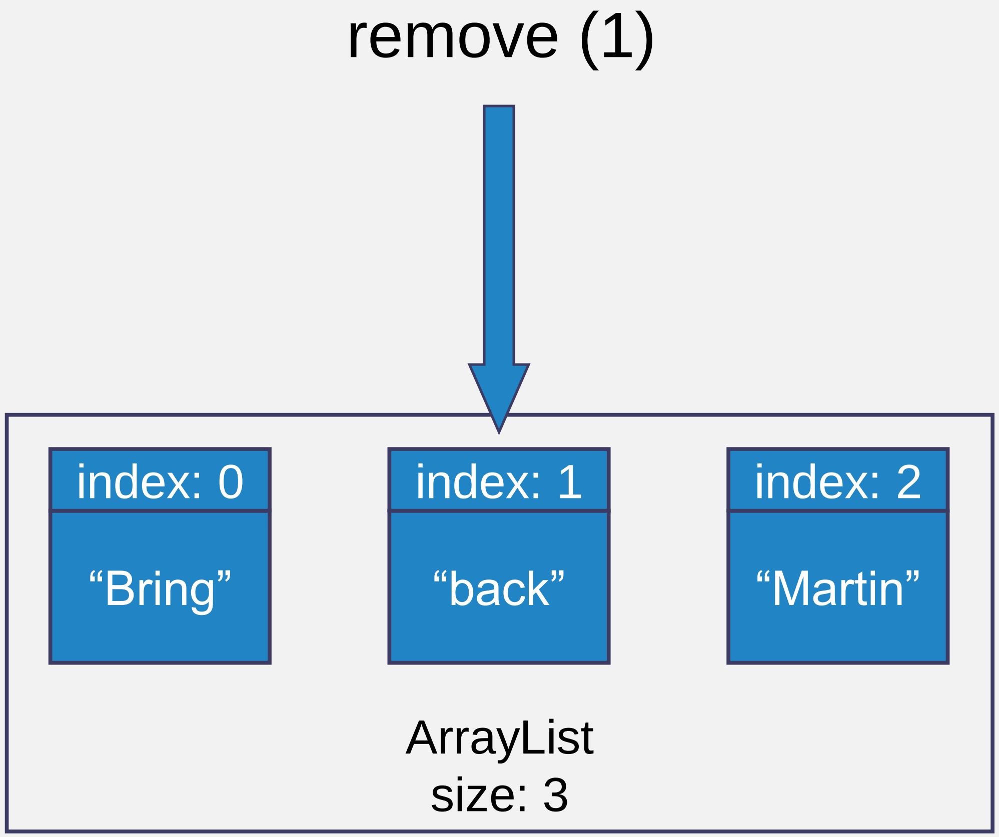
ARRAYLIST: REMOVE

Remove an element from the end of the ArrayList:



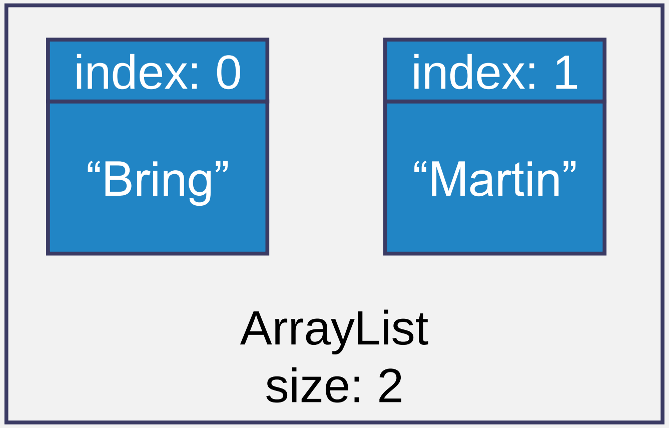
ARRAYLIST: REMOVE

Remove an element from somewhere in the middle:



ARRAYLIST: REMOVE

Remove an element from somewhere in the middle:



ARRAYLIST: CONSTRUCTING

The ArrayList class looks like this:

```
ArrayList<E>
```

One constructor looks like this (see the API for more):

```
public ArrayList<E>()
```

ARRAYLIST: CONSTRUCTING

The ArrayList class looks like this:

```
ArrayList<E>
```

One constructor looks like this (see the API for more):

```
public ArrayList<E>()
```

This looks familiar….

ARRAYLIST: CONSTRUCTING

The ArrayList class looks like this:

```
ArrayList<E>
```

One constructor looks like this (see the API for more):

```
public ArrayList<E>()
```

This looks familiar……….but what is this?

THE TYPE OF COLLECTION ELEMENTS

When creating a collection of elements, the type of those elements must be specified.

For arrays, there is a special syntax: `Type[]`

- E.g. `String[] args`

What about ArrayList?

THE TYPE OF COLLECTION ELEMENTS

When creating a collection of elements, the type of those elements must be specified.

For arrays, there is a special syntax: `Type[]`

- E.g. `String[] args`

What about ArrayList?

- Maybe a special version of ArrayList class? StringArrayList or IntegerArrayList or ObjectArrayList or etc.?

THE TYPE OF COLLECTION ELEMENTS

When creating a collection of elements, the type of those elements must be specified.

For arrays, there is a special syntax: `Type[]`

- E.g. `String[] args`

What about ArrayList?

- Maybe a special version of ArrayList class? StringArrayList or IntegerArrayList or ObjectArrayList or etc.?
- No – this would be limited to a specific number of types and not usable with new types which we create (e.g. Item, Dish, Car, Destination, GoldCoin, etc).

Similar to the ability of specifying any type of an array, there is a special syntax to do this for an ArrayList

GENERICS AND THE DIAMOND NOTATION

A type is passed to a class (classes can take parameters too!) – between the symbols '<' and '>'. This is the **diamond** notation:

`ArrayList`**`<E>`**

This allows a class (not just ArrayList) to work with elements of a provided type – the **`E`**.

The interface of the class is the same for all supplied types, providing the same services (i.e. methods) for any type of element.

This is called **generics** in Java. The function of the class is general for any type provided to the class.

ARRAYLIST: CONSTRUCTING

Back to constructing an ArrayList.

One constructor looks like this (see the API for more):

```
public ArrayList<E>()
```

ARRAYLIST: CONSTRUCTING

Back to constructing an ArrayList.

One constructor looks like this (see the API for more):

```
public ArrayList<E>()
```

Do not forget these parentheses – an ArrayList is a **class** unlike an Array. Therefore, it needs to be properly created using the standard syntax of a constructor.

E.g. `ArrayList<String> al = new ArrayList<String>();`

ARRAYLIST: EXAMPLE

Let’s define an example scenario to illustrate the concepts and syntax of ArrayLists (and other data structures) which we will discuss.

Imagine that we want to create a `ShoppingList`, which consists of a number of `Items`.

Each `Item` has a `name`, `quantity` of requested units of an item (to eventually purchase) and `price` (per unit).

We should be able to `buyMore` of an item, which increases the requested quantity by a supplied number.

ARRAYLIST: EXAMPLE

```
public class Item {
   private String name;
   private double price;
   private int quantity;

   public Item(String n, double p)
   {
       name = n;
       price = p;
       quantity = 1;
   }

   public void buyMore(int n)
   {
       quantity += n;
   }
   //getters and setters hidden
}
```

Just a standard object like we have seen so many times already: a few fields, a constructor, getters and setters, and one extra method to allow us to buy more items...

Next, we need a class to represent our shopping list

ARRAYLIST: EXAMPLE

```
public class ShoppingList {
    //Should we use an array?:
    //private Item[] itemsArray;
    //Or an ArrayList?:
    //private ArrayList<Item> itemsArrayList;

    public void addItem(Item item){
        //TODO…
    }
    //constructor and methods hidden
}
```

The choice will not affect the interface of the class, as the methods such as ‘addItem’ will handle the details of managing the collection.

However, it will affect the efficiency of writing this class and running the code overall

EXAMPLE: ARRAYLIST VS ARRAYS

“Get” syntax for retrieving the first element:

```
Item theItem = itemsArray[0];

Item theItem = itemsArrayList.get(0);
```

EXAMPLE: ARRAYLIST VS ARRAYS

“Get” syntax for retrieving the first element:

```
Item theItem = itemsArray[0];

Item theItem = itemsArrayList.get(0);
```

“Set” syntax for setting an element to the first index:

```
itemsArray[0] = new Item(…);

itemsArrayList.set(0, new Item(…));
```

EXAMPLE: ARRAYLIST VS ARRAYS

Syntax for adding an element beyond the end of the collection:

Array:

```
Item[] newArray = new Item[itemsArray.length+1];
for(int i = 0; i < itemsArray.length; i++){
    newArray[i] = itemsArray[i];
}
newArray[newArray.length-1] = new Item(…);
itemsArray = newArray;
```

EXAMPLE: ARRAYLIST VS ARRAYS

Syntax for adding an element beyond the end of the collection:

Array:

```
Item[] newArray = new Item[itemsArray.length+1];
for(int i = 0; i < itemsArray.length; i++){
    newArray[i] = itemsArray[i];
}
newArray[newArray.length-1] = new Item(…);
itemsArray = newArray;
```

ArrayList:

```
itemsArrayList.add(new Item(…));
```

EXAMPLE: ARRAYLIST VS ARRAYS

Syntax for removing an element at the end of a collection:

Array:

```
itemsArray[(totalNumberOfItems--) - 1] = null;
```

ArrayList:

```
itemsArrayList.remove(itemsArrayList.size()-1);
```

ARRAYLIST VS ARRAYS

Loops:

The syntax is very similar, just using the ArrayList specific ‘get’ and ‘set’ methods.

But be careful when using the for-each loop with an ArrayList (or any non-array collection).

Never modify the ArrayList in a for-each loop, as this leads to an error. A non-fixed length collection cannot be modified whilst iterating over it.

EXERCISE: ORDERING AN ITEM

Try the following exercise:

A shopping list contains a number of items with their associated price and quantities.

Create a method `orderFirstItem` that removes the first element in the shopping list and calculates then returns the total cost of the order, given the quantity of the item requested.

EXERCISE: SOLUTION

```
public double orderFirstItem()
 {
      Item firstItem = itemsArrayList.remove(0);
      return firstItem.getQuantity()
             * firstItem.getPrice();
 }
```

FURTHER EXERCISES

Try the following exercises in your lab session, or in your own time:

Create a method which buys a partial number of items from the list. The range of items should be supplied by the user. The bought items should be removed and the total price should be calculated and returned.

Create a method that searches for an item, by its name, within the list and buys it, which removes it from the list and returns the total price.

MORE COLLECTIONS

There are many more kinds of collections in Java’s libraries. Some of these are:

TreeSet: does not allow duplicate elements in the collection

TreeMap: maps a **key** (like an index, but of any type) to a target **value** (again, of any type)

Remember, these need to be imported from the **java.util** library into your program just like the ArrayList. E.g.:

```
import java.util.TreeSet
import java.util.TreeMap
```

SET AND TREESET

Another kind of Collection offered by the Java libraries is a **TreeSet**. This is one variation of a **Set** (more on this in future).

The key difference between a Set and other kinds of collections is that it cannot contain duplicate elements.

A Set **is not** the same as a List in Java.

SET AND TREESET

Another kind of Collection offered by the Java libraries is a **TreeSet**. This is one variation of a **Set** (more on this in future).

The key difference between a Set and other kinds of collections is that it cannot contain duplicate elements.

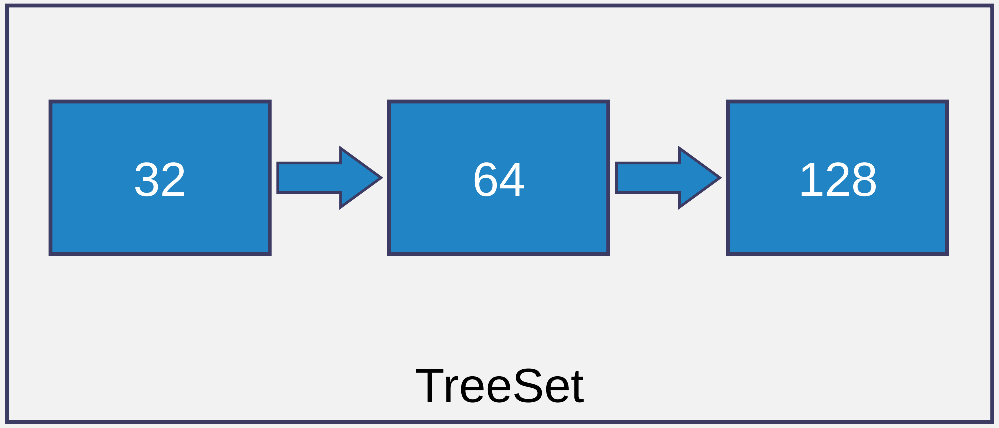
A Set **is not** the same as a List in Java. Sets do not have indexes.

A TreeSet **orders** the elements of a set according to their natural or specified ordering.

SET AND TREESET

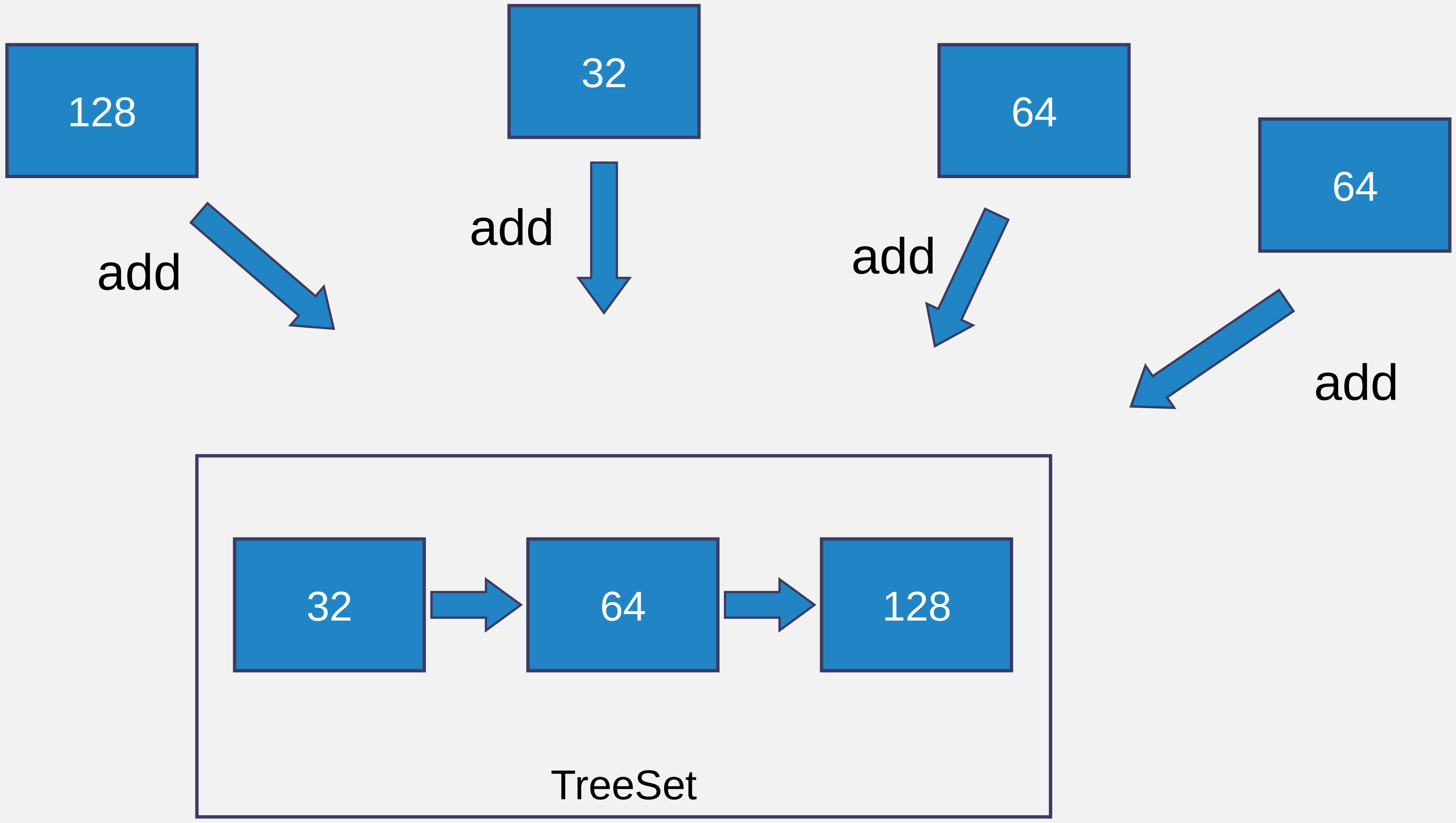
TreeSet containing integers:

```
TreeSet<Integer> numSet = new TreeSet<Integer>();
numSet.add(128);
numSet.add(32);
numSet.add(64);
numSet.add(64);
```



TREESET

TreeSet containing integers:



TREESET: EXAMPLE CODE

Imagine we want to be able to buy all the items in the shopping list, implemented as a TreeSet.

```
public class SetTest {
private TreeSet<Item> items = new TreeSet<Item>();
//avoid this, initialise in constructor
 public double buyAllItems()
 {
         double totalPrice = 0;
         for(Item item : items){
                 totalPrice += item.getPrice() * item.getQuantity();
         }
         items.clear();
         return totalPrice;
 }
}
```

(TREE)SET VS (ARRAY)LIST

Key differences:

ArrayList is a List, which uses indexes.
TreeSet is a Set, which does not use indexes.

ArrayList is a List, which can contain duplicate elements.
TreeSet is a Set, which cannot contain duplicate elements.

In this context, the prefix **Tree** indicates an ordering of the elements in the collection.

MAP AND TREEMAP

Another kind of Collection offered by the Java libraries is a **TreeMap**. This is one variation of a **Map** (more on this in future).

A Map is a (key, value) pair, where a key maps to a specific value. It is a one-to-one mapping.

MAP AND TREEMAP

An Array maps integer keys – indexes – to values.

```
String[] names = new String[4];
names[0] = "Martin";
names[1] = "Asad";
names[2] = "Steffen";
names[3] = "Tomas";
```

names

int	String
0 →	→ "Martin"
1 →	→ "Asad"
2 →	→ "Steffen"
3 →	→ "Tomas"

MAP AND TREEMAP

Using Maps, we can use indexes other than just integers

itemPrices

String	Double
Chocolate	1.60
Bananas	0.30
Apples	0.20
Laptop	499.99
…	…

MAP AND TREEMAP

A **TreeMap** orders the elements in the map by a natural or specified ordering of the **keys**.

E.g. alphabetical ordering:

```
TreeMap<String, Double> itemPrices
    = new TreeMap<String, Double>();
itemPrices.put(“Laptop”, 499.99);
itemPrices.put(“Apples”, 0.20);
itemPrices.put(“Chocolate”, 1.60);
itemPrices.put(“Bananas, 0.30);
```

itemPrices

String	Double
Apples	0.20
Bananas	0.30
Chocolate	1.60
Laptop	499.99
…	…

TREEMAP: SHOPPING LIST

```
public class ShoppingList {
 private TreeMap<String,Item> items;
 public ShoppingList()
 {
        items = new TreeMap<String,Item>();
 }
 public void addItem(Item i)
 {
        items.put(i.getName(),i);
 }
 public void addExtra(String s,int n)
 {
        items.get(s).buyMore(n);
 }
 public void drop(String s)
 {
        items.remove(s);
 }
 public Item get(String s)
 {
        return items.get(s);
 }
}
```

We can use a Map to index our ShoppingList with the String names of the Items

The TreeMap links Strings to Items

Adding an item now uses *put* and requires the name (the index) and the value (the item)

Look how easy it is to get hold of items now!
We no longer need loops

DATA STRUCTURES: LESSONS

Some lessons to observe:

See how the implementation of a ShoppingList changed?

- We used an array of Items – familiar, but not very convenient
- Then used ArrayList<Item> - easy to use
- Finally used a Map<String,Item> - slightly more complicated, but some advantages…

But, the interface of our class did not change

- This is an example of why *encapsulation* (making the implementation private) matters
- All the changes to the implementation are invisible to the user

However, user can see a difference in performance

- Compare ArrayLists and TreeMaps on a Big Shop….

AFTER THOUGHTS

We have left a few mysteries for later:

- Why is a TreeSet a Set, or a TreeMap a Map? What does this mean?
- Why did we not use or instantiate a List or Map? Turns out that we can’t, but why?

More sophisticared things we will not talk about, but that might interest you:

- You can build generic classes of your own, and also generic methods.
- You can also restrict generics to work with specific types that meet specific requirements.

Topic 8: Data Structures

Programming Practice and Applications (4CCS1PPA)

Dr. Asad Ali
Thursday 24th November
programming@kcl.ac.uk