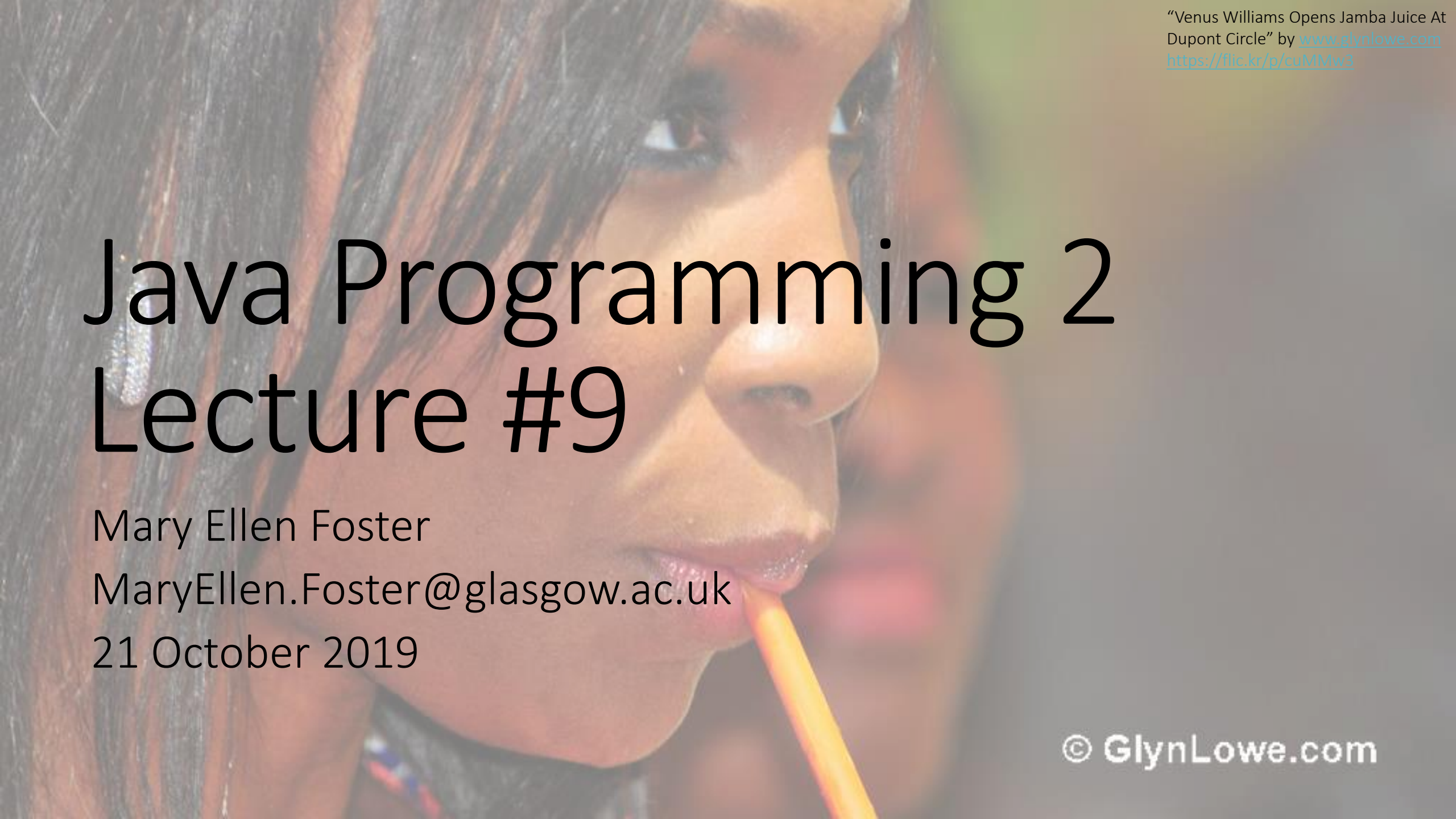
Find the bug in this code …

```
public int findMinValue(int[] ints) {
    int min = 0;
    for (int i : ints) {
        if (i < min) {
            min = i;
        }
    }
    return min;
}
```



"Venus Williams Opens Jamba Juice At Dupont Circle" by www.glynlowe.com https://flic.kr/p/cuMMw3

Java Programming 2 Lecture #9

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk

21 October 2019

Outline

Java packages

Importing package members

Packages and importing in Eclipse

Introduction to the Java Collections framework

Packages

Package: groups together related resources (usually classes)

Why put code in a package?

- Makes it obvious that types are related
- Makes it possible to find types for a specific purpose (given good package name)
- Type names won't conflict with names from other packages
- Types within package can have unrestricted access to one another
  - *While restricting access for types outside the package*

Visibility modifiers (revisited)

Modifier	Same class	Same package	Any subclass	Any class
`public`	●	●	●	●
`protected`	●	●	●	
(default)	●	●		
`private`	●			

Used to limit the visibility of class members (fields and methods)

Specify as part of member declaration – `private int balance;`

Creating a package

Choose a name (details on naming scheme next)

Put a `package` statement at the top of every source file for that package

```
package my.package.name;
```

Ensure that all source files are in a directory corresponding to the package name

If you don't use a `package` statement, then all files are in the default package

Package naming conventions

Rules are the same as java identifiers

- Can't start with a digit; can't contain special characters or hyphens; can't contain a reserved keyword such as `int` or `new`
- Components separated by period "."

Conventions

- All lower case
- Built-in packages start with `java.` or `javax.`
- Companies and organisations usually use their domain name, reversed:
  - `com.example.mypackage` – *a package named* `mypackage` *from a programmer at* ***example.com***
  - `uk.ac.glasgow.dcs.jp2` – *possible package for code for this class*

Accessing package members

To use a public type from outside the package, do one of:

- Refer to it by its fully qualified name
- Import the member itself
- Import the entire package

Note: `java.lang` is always available by default with no special effort

- `String`, `Double`, `Exception`, …

Other packages are imported by default in JShell for convenience

- **`java.io`, `java.math`, `java.net`, `java.nio.file`, `java.util`**, and lots of **`java.util`** subpackages
- Just about any packages we are using during this course, actually!

Using fully qualified name

```
java.util.Scanner stdin = new java.util.Scanner(System.in);
```

Works well for infrequent use

Code can easily become repetitive and hard to read

Importing a single member

```
// At top of source file, after package statement
import java.util.Scanner;
// … later on, inside the class …
Scanner stdin = new Scanner(System.in);
```

Works well if you use a few members from a package

Can get a bit silly if you use many types from the same package

Importing a package

```
// At top of source file, after package statement
import java.util.*;
// … later on, inside the class …
Scanner stdin = new Scanner(System.in);
```

Useful if you need lots of members from the same package

Use is controversial:

http://stackoverflow.com/questions/147454/why-is-using-a-wild-card-with-a-java-import-statement-bad

More on package names

Package names look like they *might* be a hierarchy

- `java`
  - `java.awt`
    - `java.awt.color`
    - `java.awt.font`
    - …

But they are **not**!

- `import java.awt.*` does **not** import any classes from `java.awt.color` or anywhere else

File names and directories

Source code for a class should be in a file corresponding to the class name

```
public class CreditCard { … }
```
CreditCard.java

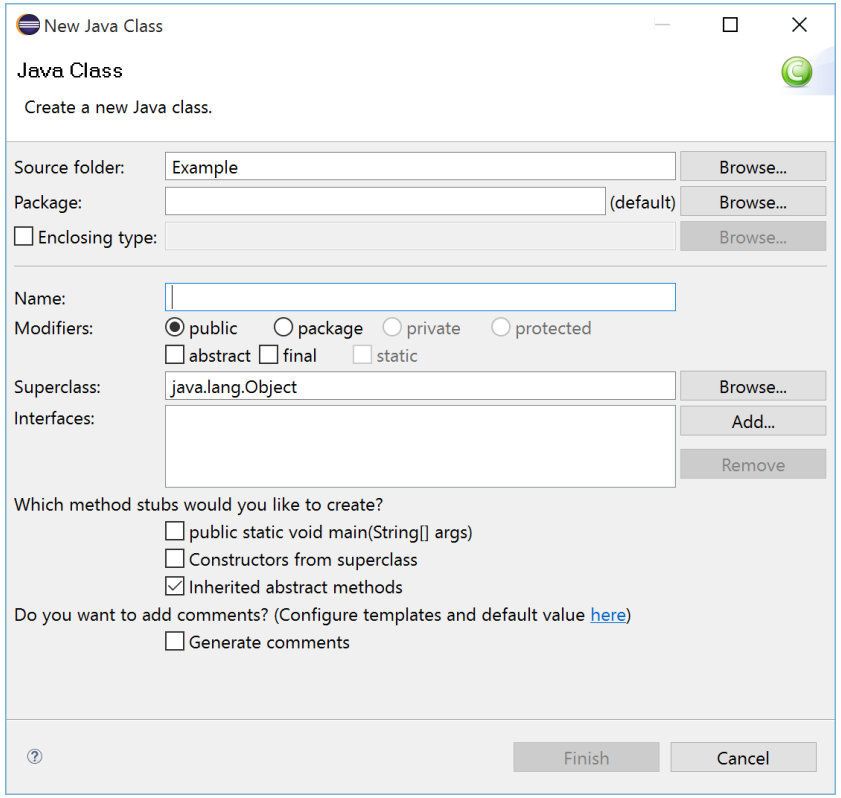
Package determines the directory that the file should be in

```
package uk.ac.glasgow.dcs.jp2;
public class CreditCard { … }
```
…\uk\ac\glasgow\dcs\jp2\CreditCard.java

All paths are relative to the current working directory

Working with packages in Eclipse

- "Package explorer" view – useful when you go beyond the default package
- Creating a new package:
  - Right-click project -> New -> Package
- Creating a class in a package
  - Right-click project -> New -> Class
  - Specify package – it will be created if it does not already exist
- Moving class to a new package
  - Right-click class -> Refactor -> Move



Managing imports in Eclipse

Essential keyboard shortcuts:

Ctrl-<space> on (partial) class name*

Pops up class-name autocompletion

Once you choose a class, it automatically adds the necessary `import` statement

Ctrl-Shift-O (letter “o”, not number “zero”)

Organises imports

Removes unused ones

Sorts the rest nicely

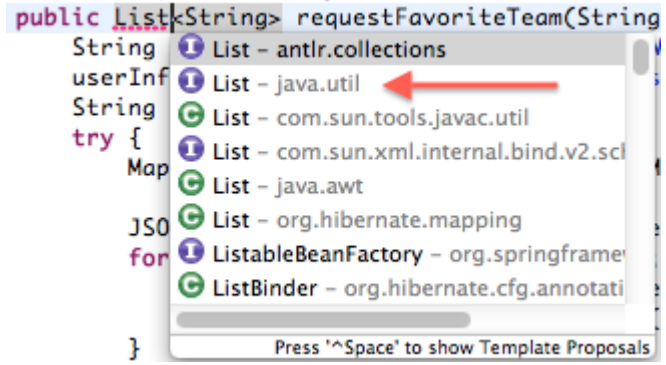


Image from https://mihail.stoynov.com/2011/08/24/longstanding-eclipse- issues-fix-them-finally-please/

* Ctrl-<space> also works to autocomplete many other things – fields, method names, variables, exceptions, even whole methods (try “mai Ctrl-<space>”). Try it out and see!

Arrays revisited

Why use an array?

- Storing a group of values
- Directly supported by underlying Java Virtual Machine (JVM) – **efficient**

Major limitation: **fixed size**

- Size is determined when array is created

```
int[] numbers = new int[10];
String[] strings = { "each", "peach", "pear", "plum" };
```

- If you go past the end, you get an `ArrayIndexOutOfBoundsException`
- If you don't use all the space, you have wasted the additional capacity

Java Collections framework

A standard set of built-in classes for representing and manipulating collections

Each Collections class groups **related elements** into a **single unit**

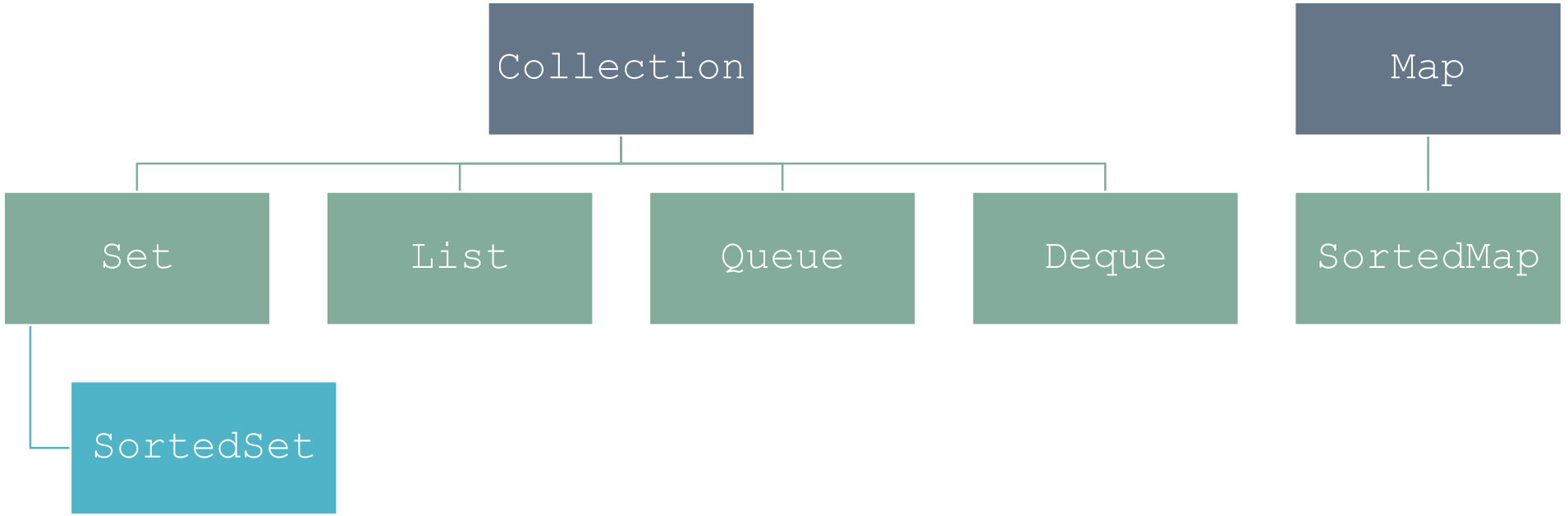
Examples:

- `ArrayList` – acts like a variable-length array
- `HashSet` – a group of unique elements
- `Stack` – a list with last-in/first-out semantics
- `HashMap` – a **dictionary** (e.g., a telephone directory)

Structure of Collections framework

Base class:* `java.util.Collection`

Methods: `add(), remove(), contains(), size(), toArray()`



** Actually, everything in this picture is technically an interface – we will discuss the details next week.*

Advantages of Collections

Reduces programming effort by providing pre-written data structures and algorithms

Increases performance by providing high-performance implementations
    Implementations are interchangeable – can switch to tune performance

Provides interoperability by allowing Collections to be passed back and forth

Reduces effort to learn new APIs by providing a common interface

Reduces effort to design APIs by giving design specifications

Fosters software reuse by providing a standard interface

(List adapted from https://docs.oracle.com/javase/tutorial/collections/intro/index.html)

java.util.ArrayList

A Collections class (specifically, a `List`) that implements **variable-length** arrays

More flexible than built-in arrays, but less efficient

Acts as a wrapper around an underlying array that grows and shrinks dynamically

`ArrayList` is a **class** – so elements are added and removed by **methods**
(Not by built-in Java syntax as with normal arrays)

It has a **capacity** (size of internal array) and a **size** (number of elements in the list)

Capacity is increased when necessary – purely internal

Size is increased/decreased as elements are added and removed, and checked for operations

In general: IndexOutOfBoundsException if `(index < 0 || index >= size())`

Creating an ArrayList

```
// Default initial capacity (10)
ArrayList<String> strings = new ArrayList<>();

// Explicit initial capacity
ArrayList<String> strings = new ArrayList<>(50);
```

Size and capacity

```
// Returns size
int size = strings.size();

// Checks whether list is empty (i.e., is size == 0)
if (! strings.isEmpty() ) { … }

// Trims capacity to current size
strings.trimToSize();

// Ensures minimum capacity
strings.ensureCapacity(100);
```

Adding elements

```
// Adds the element to the end of the list
// Always succeeds; increases capacity and/or size as necessary
strings.add ("foo");


// Adds the element at the given index, and shifts other elements
// May throw IndexOutOfBoundsException
strings.add (5, "foo");


// Sets the element at the given index to the new value
// May throw IndexOutOfBoundsException
strings.set (5, "foo");
```

Accessing and removing elements

```
// Returns element at the given position
// May throw IndexOutOfBoundsException
String s = strings.get(5);


// Removes (and returns) element at the given position
// Shifts all remaining elements to the left
// May throw IndexOutOfBoundsException
String s = strings.remove(5);


// Removes first occurrence of given element in list
// Shifts all remaining elements to the left
// Returns true if element was there, and false if not
if (strings.remove ("foo") ) { … }
```

Checking list contents

```
// Returns true if the list contains the given element
if (s.contains (“foo”)) { … }


// Returns index of first occurrence of element in list
// (or -1 if it’s not there)
int i = strings.indexOf (“foo”);


// Returns index of last occurrence of element in list
// (or -1 if it’s not there)
int i = strings.lastIndexOf (“foo”);
```

Array vs ArrayList at a glance

Operation	Array	ArrayList
Declaration	`String[] strings;`	`ArrayList<String> strings;`
Initialisation	`strings = new String[10];`	`strings = new ArrayList<>(10);`
Setting element	`strings[5] = "foo";`	`strings.set(5, "foo");`
Accessing element	`String s = strings[5];`	`String s = strings.get(5);`
Getting size	`int n = strings.length;`	`int n = strings.size();`
Adding element	*n/a*	`strings.add("foo");` `strings.add(5, "foo");`
Removing element	*n/a*	`strings.remove("foo");` `strings.remove(5);`

Finding element	`Arrays.binarySearch(strings, "foo");`	`strings.contains("foo");`
		`strings.indexOf("foo");`
		`strings.lastIndexOf("foo");`

Next time

More on Collections framework and `ArrayList`

Iterating over Collections

Generic types

Other useful Collection types

Feedback on Python/Java quiz