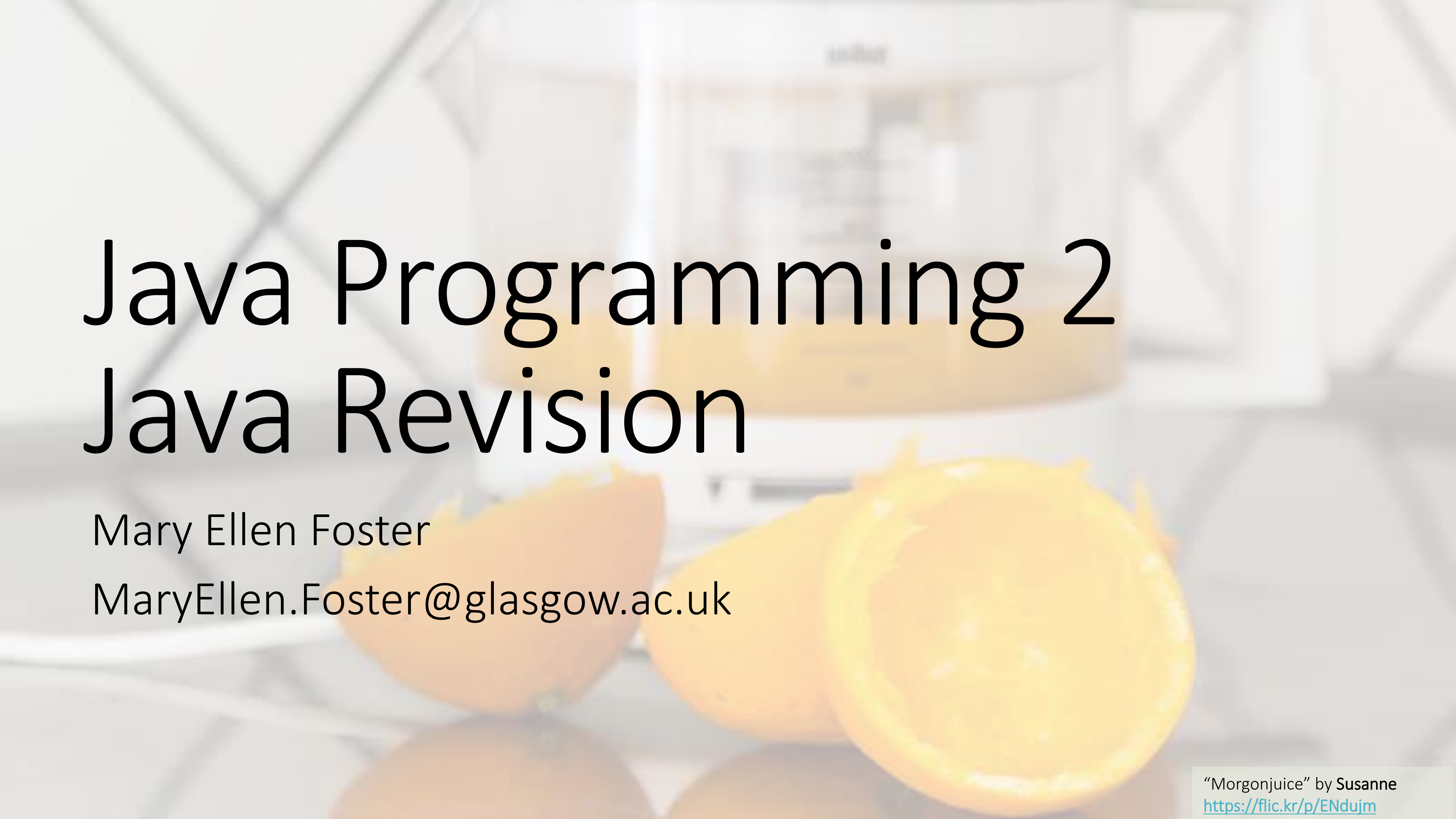

Java Programming 2
Java Revision

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

“Morgonjuice” by Susanne
https://flic.kr/p/ENdujm

Primitive types and identifiers

Primitive types: `byte, short, int, long, float, double, boolean, char`

- Corresponding wrapper classes (`Byte, Short`, etc – don't forget `Integer` and `Character`)
- Converting between primitive types and wrappers: **boxing** and **unboxing**

Identifier: label for a named Java entity (class, field, method, parameter, variable, ...)

- **Rules:** begin with letter or underscore, continue with letter/number/underscore
- **Conventions:** Classes start with capital, other things with lower case, use camelCase, constants in ALL_CAPS

Type conversions

Java is **statically typed** – any variable can only hold values of a single, specific type

To store a value of type t_1 in a variable of type t_2, the value must be **converted to** t_2 **before** the assignment occurs

- Implicit conversions: happen automatically, (little or) no information lost
  - *Byte -> long, int -> double, subclass -> superclass*
- Explicit conversions (casting): must be explicitly signalled due to potential info loss
  - *Double -> float, int -> byte, Object -> specific class*
  - *May cause ClassCastException at runtime if cast is not valid*
  - *Details of how narrowing works in practice*

Built-in methods for converting String <-> primitive types

- String.valueOf(), Integer.parseInt()

Arithmetic operations, integer division

Arithmetic operators: +, -, *, /, %

Function of division operator "/" depends on type of the two arguments

- If **both** are integers (`int, long, short, byte`), then it does **integer division**
- If **either** is floating-point (`float, double`), then it does **floating point division**

Example:

`7.0/4.0`	returns **`1.75`** (same result for `7.0/4` and `7/4.0`)
`7/4`	returns **`1`**

General rule: integer division throws away the remainder (so `99/100 == 0`)

Control flow

- `for` and `while` loops
  - Condition is checked each time the loop is started; entire loop executed before condition is checked again
    - *Skip the rest of the current loop execution:* **`continue`**
    - *Terminate the loop immediately:* **`break`**
- `for-each` loops: more efficient method of iterating through arrays or collections
- `if` statements vs `switch` statements
  - Switch evaluates an integer or String (or enum constant), and executes one or more `case` blocks
  - Don’t forget to include `break`

Objects, classes, inheritance

Characteristics of objects: **state**, **behaviour**

An object is an **instance** of a general **class** of objects

In Java, a class contains **fields** (state) and **methods** (behaviour)
- **Static** fields/methods are associated with the class itself, not with an instance

Classes can **inherit** state and behaviour from other classes
- Subclass is a **specialised version** of the superclass

In Java, a class can have **exactly one** superclass
- If superclass isn't specified, then it inherits from `Object`

Subclasses can **override** superclass methods to provide specialised behaviour

Don't forget **access modifiers** (`public/protected/default/private`)

More on OO concepts

Constructor: used to create a new instance of a class (via **new** keyword)

- Constructors are **not** inherited – call super-class constructor with **super** keyword

Method **overriding**: redefining method behaviour in a subclass

Method **overloading:** defining multiple methods with the same name but different signatures

Details of Java methods

A method declaration has six components (in order):

1. Access modifier(s) (zero or more)
2. Return type (`void` if it does not return a value)
3. Method name (conventionally beginning with a verb)
4. Parameter list in parentheses – comma delimited list of input parameters, preceded by data type, enclosed in parens. No parameters – empty parens.

   Method signature
5. An exception (possibly empty)
6. The method body, enclosed in braces { }

Abstract classes/methods, interfaces

Abstract classes have “holes” – abstract methods that **must** be overridden

- Still have constructors, fields, normal methods, static fields/methods, etc

Final classes cannot be subclassed (e.g., for security), and final methods cannot be overridden

- **Final** fields, parameters, variables cannot have value changed after it is set
- **Static final** generally indicates class-level constants (e.g., `Long.MAX_VALUE`)

Interfaces represent class relationships **outside main inheritance hierarchy**

- Classes **implement** interfaces – can implement any number of them (including zero)
- All methods implicitly **public abstract**; all fields **public static final**
- Support multiple inheritance of **type** (not of state or of implementation)

Exceptions

When an error occurs in program execution, an `Exception` is **thrown**

Unless the exception is **caught**, the entire program will crash

Checked exceptions **must** be caught; unchecked exceptions may be ignored (but will still crash program if thrown)

Exception handling options

1. *Try/catch – deal with the exception where it happens*
2. *Re-throw – inform calling code that it needs to address the exception (add **throws** to signature)*

Advantages of using Exceptions:

Separates error-handling code; propagates errors to a method that can handle them; groups errors into types (Exception is a class and can be subclassed)

In general, throwing an Exception as part of the core control flow is considered bad style

Packages

Group together related resources (usually classes)
- Make it obvious types are related, reduces naming conflicts

Put package statement at top of every source file in the package:

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

If you don't use a package then all files are in default package

Packaging interacts with visibility modifiers (specifically protected vs default)

Using code from a different package:
- Use fully qualified name everywhere (java.util.ArrayList) – discouraged
- Import the package at the top of the source file and just use class name

Arrays, Collections, Generics

Arrays: fixed length sequence of consecutive memory locations (efficient to use)
- Has a **type**: specifies element type and dimensionality (`int[]`, `String[][]`)

Collections: set of built-in classes for representing and manipulating collections
- **List** – acts as a variable-length array
- **Set** – unordered collection
- **Map** – dictionary type

Above are all **interfaces** – to create a concrete object, use, e.g., ArrayList / HashSet / HashMap

Iterating through an array or a collection: use **for-each** loop

Converting between array and Collection: use `java.util.Arrays` class (useful set of static methods) and toArray() method

All collections are **generic** – includes type param ArrayList<String>
- Provide strong type check at **compile time** (instead of weird errors at **run time**)

File input/output with java.nio

Basic concept: **Path** (identifies a location in the file system – which may not exist!)

Lots of methods for manipulating Paths

Use Files class (static methods) for manipulating actual files/directories

- Most methods work on Path instances

equals, hashCode(), Comparable

- equals() method – defines when two objects are considered equal
  - Default implementation: returns whether they are the **same** object (via ==)
  - Important: signature must be **boolean equals (Object obj)**
  - Use Eclipse to auto-generate, or else use **Objects.equals()**
- hashCode() – returns an int corresponding to the object
  - **Should be overridden whenever equals() is overridden**
- Comparable<T>: generic interface used to define an **ordering** on objects
  - One method: **public boolean compareTo (T t)**
  - Does not have to agree with **equals()** (but it is good practice)

GUI programming with Swing

Swing uses modified Model-View-Controller – View+Controller = “UI Delegate”

Basic programming strategy

- Create top-level container (e.g., JFrame)
- Create necessary models for components that need them (list, table, etc)
- Create GUI elements and add them to container (button, table, etc)
- Set up the container layout
- Add event-handling code (listeners)
- Display window on screen and wait for user interaction

Threads

Concurrent programming: multiple things happening at once

- Benefit: execute subtasks in parallel for efficiency
- Costs: Threads can access shared data – problems include visibility (thread B changes data without thread A's knowledge) and access (several threads access and change data at same time)

Creating a Thread: implement **Runnable** interface and define **run()** method

- Thread methods: **start()**, **sleep()**, **join()**, **interrupt()**

Avoiding thread interference: impose an ordering (**happens-before** relationship)

Keyword to impose ordering: **synchronized**

Threads continued; immutable classes

Atomic access: effectively happens at once, cannot be interrupted

Liveness problems: deadlock, starvation, livelock

Immutable: internal object state cannot change after it is constructed (e.g., String)

- Can be safely shared, used for lookup in dictionary-type structures
- Recall with String: methods either access state or **return a new modified String** (e.g., toUpperCase(), trim())

Higher-level concurrency: Lock and condition objects

Functional programming with streams

All Collection objects can be converted to a java.util.stream.Stream

- Represents a sequence of values
- Exposes a set of **aggregate operations**
- All intermediate operations return a new Stream to allow operations to be **chained**
- Terminal operations produce a single result and/or a side-effect (**forEach**)
    - *Result may be Optional – must check for isPresent() or use orElse()*

Powerful but sometimes tricky to use

Enumerations

Enum: special data type that allows a variable to be one of a set of constants

- Examples: days of week, 22-point grading scale, compass directions, ...

Declared with **enum** keyword instead of **class**

Can also have fields, constructors, other methods, etc

Can be compared with ==

Can be used in switch statements

Can access names and ordinal positions

Annotations

Provide **metadata** about a program

Uses:

- Information for the compiler – detect errors, suppress warnings
- Compile-time processing – generate code/XML/etc

Examples:

- @Override
- @SuppressWarnings
- @Test, @Before, @After (JUnit)

Programming style

“Always code as if the [person] who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.”

John F. Woods

Javadoc (and other) comments

Annotation

Indentation

Variable naming

Appropriate declarations – e.g., List<> vs ArrayList<>

Returning values appropriately

…

Other topics I didn't cover

Inner and anonymous classes
- https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

Lambdas
- https://www.baeldung.com/java-8-lambda-expressions-tips

Modules
- https://www.oracle.com/corporate/features/understanding-java-9-modules.html

Reflection
- https://docs.oracle.com/javase/tutorial/reflect/index.html

Date and Time APIs
- https://docs.oracle.com/javase/tutorial/datetime/index.html