Java Programming 2 Abstract classes, final classes and members

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

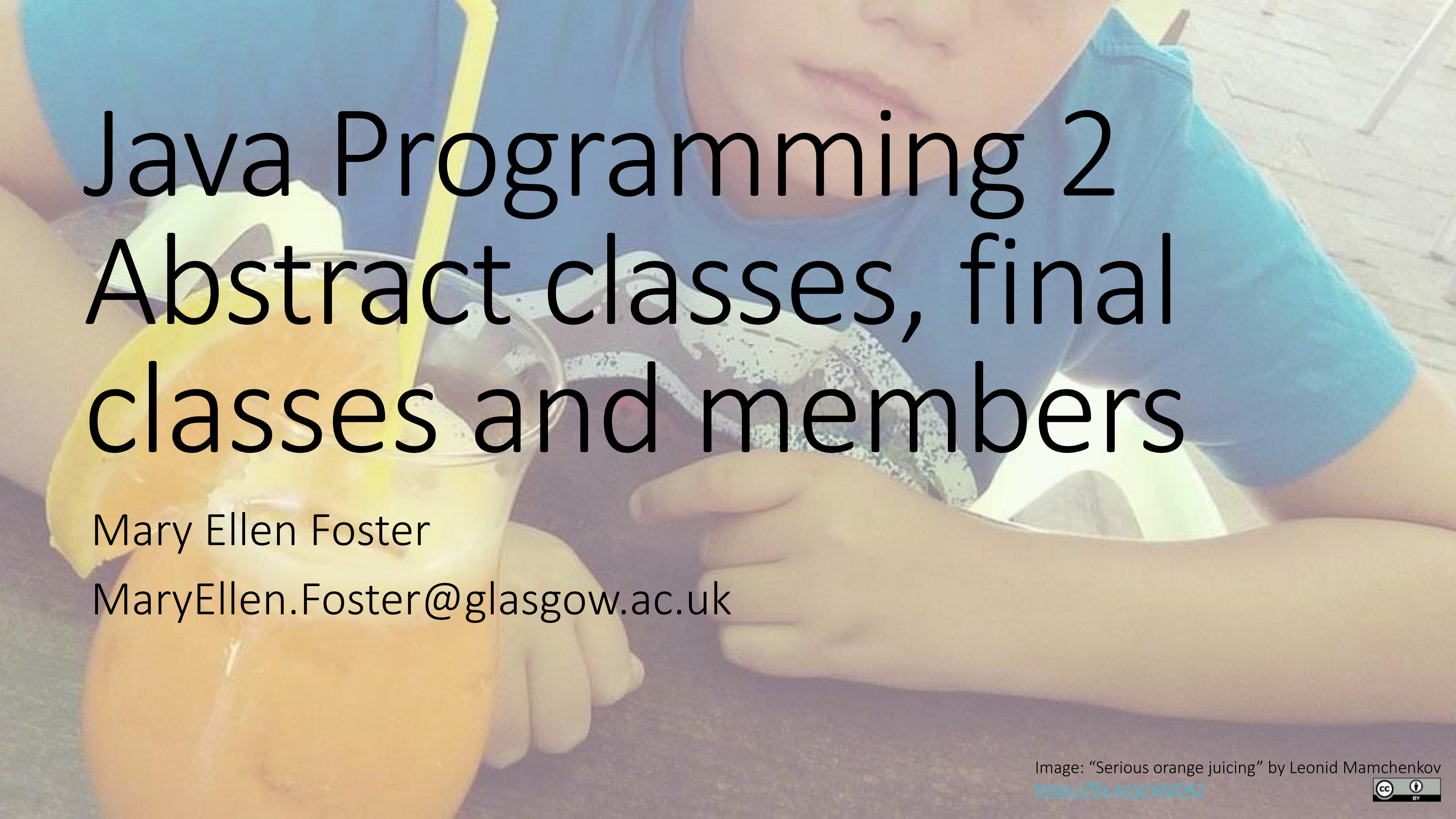


Image: “Serious orange juicing” by Leonid Mamchenkov
https://flic.kr/p/xXdD42

Overriding methods

Basic process

- Method is defined in superclass
- Method is redefined in subclass

Then Java can assume that every object of that type, whatever the subclass, can provide that behaviour (polymorphism)

But what if there is no sensible implementation in the superclass?

```
public class Animal {
    public void move() {
        System.out.println(“animals can move”);
    }
}

public class Dog extends Animal {
    public void move() {
        System.out.println(“dogs can walk and run”);
    }
}
```

Abstract classes and methods

Some classes have “holes” in them – methods that **must** be overridden in subclasses

Such classes are marked as `abstract`

Methods that must be overridden are marked as `abstract` too

If a subclass does not implement all `abstract` methods, it must also be marked `abstract`

First abstract watercolor, painted by Wassily Kandinsky, 1910.

Example

```
public abstract class TwoDimensionalPoint {
  protected double x;
  protected double y;

  public abstract double distanceToOrigin();
}

 public class CartesianPoint extends TwoDimensionalPoint {

   public double distanceToOrigin() {
     return Math.sqrt(x*x+y*y);
   }
 }

 public class ManhattanPoint extends TwoDimensionalPoint {

   public double distanceToOrigin() {
     return Math.abs(x) + Math.abs(y);
   }
 }
```

This method ensures that all subclasses meet a given API

- In the example, all subclasses of `TwoDimensionalPoint` must implement `distanceToOrigin()`

But: it doesn’t make sense to implement `distanceToOrigin()` in the superclass

More on abstract methods/classes

Abstract methods do not have a body – just the signature followed by semicolon

```
public abstract double distanceToOrigin();
```

Abstract classes can still have

- Constructors
- Fields
- Normal (non-abstract) methods
- Static fields and methods

You **cannot** create instances of abstract classes – only **concrete** subclasses

(Opposite of abstract)

~~`TwoDimensionalPoint p= new TwoDimensionalPoint();`~~

Inheritance issues

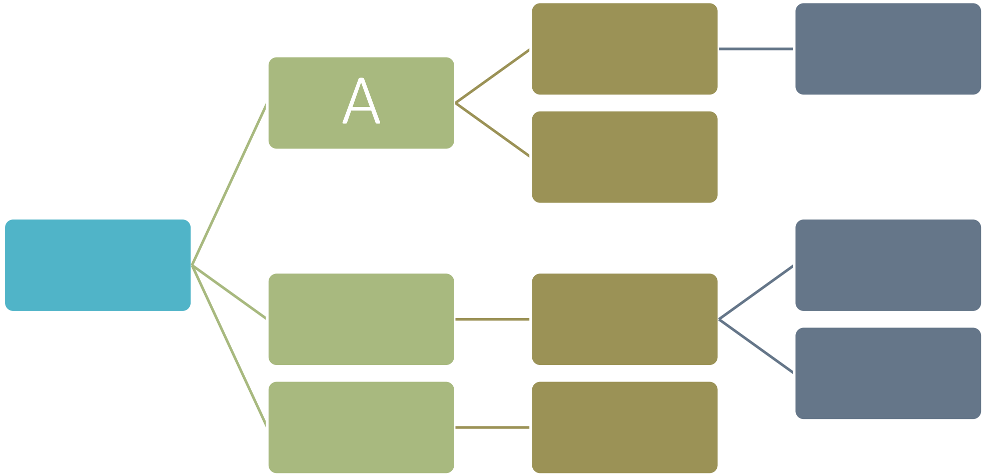
Recall: **polymorphism** means that, if code is expecting an instance of class A, you could use an instance of any subclass of A

- Subclass might override any of A's methods with its own implementation

What if A has a critical function?

- Checking passwords
- Accessing a critical piece of hardware
- …

Subclass injection attack



Example

```
public class PasswordChecker {

  public boolean check(String username, String password) {

    String passwordHash = hash(password);
    String correctHash = lookupHash(username);
    return (passwordHash.equals(correctHash));

  }

}


public class DodgyChecker extends PasswordChecker {

  public boolean check(String username, String password) {

    return true;

  }

}
```

Solution: the `final` keyword

If a method is marked as `final` then it **cannot be overridden**

- Provides predictable behaviour
- Especially relevant where method has security implications

If a class is marked as `final` then it **cannot be subclassed**

- Particularly useful for **immutable** classes such as `String` or `Double`
- … Or if all methods would require `final`

Improved password checker

```
public final class PasswordChecker {

  public boolean check(String username, String password) {

    String passwordHash = hash(password);
    String correctHash = lookupHash(username);
    return (passwordHash.equals(correctHash));

  }

}
```

or

```
public class PasswordChecker {

  public final boolean check(String username, String password) {

    String passwordHash = hash(password);
    String correctHash = lookupHash(username);
    return (passwordHash.equals(correctHash));

  }

}
```

`final` fields, parameters, and variables

If a **field** is declared `final`, then its value can never be changed

- Value can only be set at declaration time or in a constructor

If a **parameter** is declared `final`, then its value can never be changed inside the method

If a **variable** is declared `final`, then its value can never be changed

- Value can be set at declaration or later, but can never be changed thereafter

```
public class Test {
        private final int field1 = 1;
        private final int field2;

        public Test (final int arg) {
                this.field2 = arg; // okay
                this.field1 = 5;   // error

                arg = 3;           // error

                final int foo;     // okay
                final int bar = 2; // okay

                foo = 3;           // okay
                foo = 4;           // error
                bar = 4;           // error
        }
}
```

What about `static final`?

Generally used to define **constants**

- `final` modifier means that the value cannot change
- Constant names are (usually) written in `ALL_CAPS`

Examples:

- `Math.E` *The double value that is closer than any other to e, the base of the natural logarithms*
- `Long.MAX_VALUE` *A constant holding the maximum value a* `long` *can have,* $2^{63}-1$
- `System.out` *The "standard" output stream*