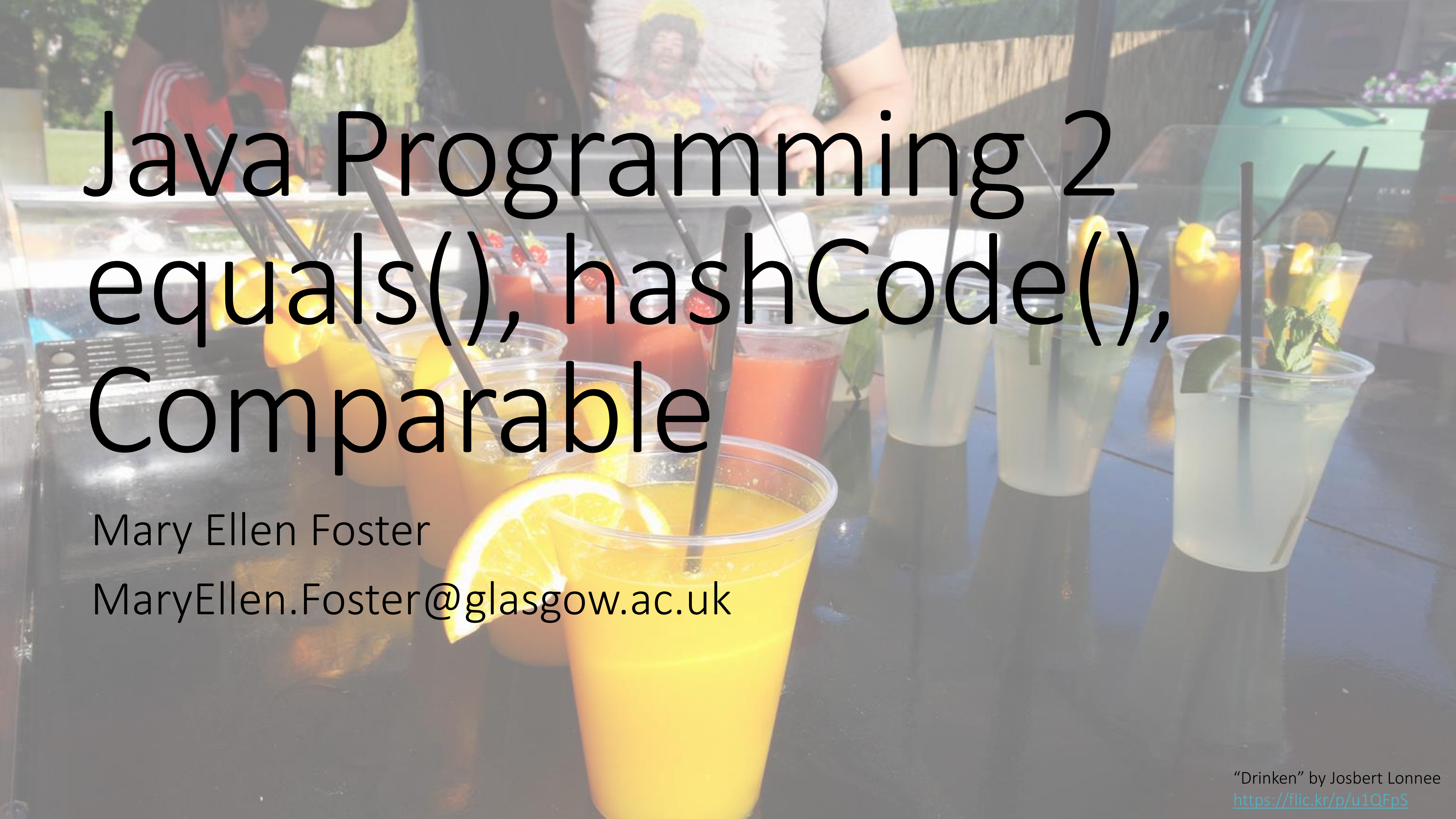


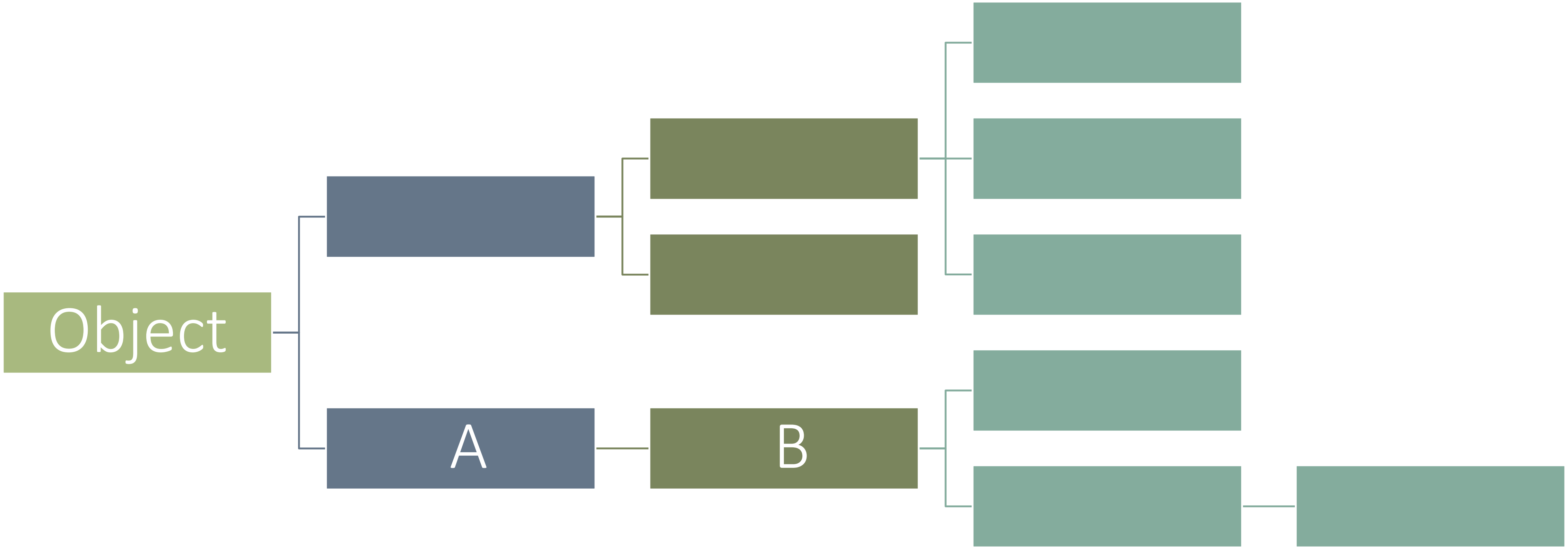
Java Programming 2 equals(), hashCode(), Comparable

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

“Drinken” by Josbert Lonnee
https://flic.kr/p/u1QFpS

Inheritance in Java



Methods of `java.lang.Object`

`protected Object clone()`

`boolean equals (Object obj)`

`protected void finalize()`

`public Class<?> getClass()`

`public int hashCode()`

`public void notify() / notifyAll()`

`public String toString()`

`public void wait() / wait(long timeout) / wait(long timeout, int nanos)`

java.lang.Object.equals() documentation

Indicates whether some other object is "equal to" this one

The `equals` method implements an equivalence relation on non-null object references:

- It is ***reflexive***: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is ***symmetric***: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is ***transitive***: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is ***consistent***: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

Default implementation of equals()

"The most discriminating possible equivalence relation on objects"

Returns true **if and only if** x and y refer to the **same** object (i.e., x == y is true)

Gives the correct result for primitive types (int, double, char, etc.)

Does not check if objects are **equivalent** – i.e., if their contents are the same

```
ArrayList<Integer> l1 = new ArrayList<>();
l1.add (1);
ArrayList<Integer> l2 = new ArrayList<>();
l2.add (1);
boolean result = l1.equals(l2); // Default would return false
```

java.lang.Object.hashCode() documentation

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Default implementation

“As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)”

equals() and hashCode()

"If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects **must produce the same integer result.**"

So if you override `equals()`, **you must also override `hashCode()`**

Overriding equals() and hashCode()

```java
public boolean equals(Object obj) {
  if (obj == this) return true;
  if (obj instanceof Song) {
    Song s = (Song) obj;
    return Objects.equals(s.artist, this.artist) && Objects.equals(s.title, this.title);
  }
  return false;
}

public int hashCode() {
    return Objects.hash(artist, title);
}
```

```java
public class Song {
    private String artist;
    private String title;

    // …
}
```

The `Comparable` interface

Built-in interface that declares how objects are compared to one another for sorting
- If a class implements `Comparable`, then lists of that type can be sorted

Defines a `compareTo` method which returns:
- < 0 if this object is “less than” the other one
- > 0 if this object is “greater than” the other one
- = 0 if this object is “equal to” the other one

```
package java.lang;

public interface Comparable<T> {
    public int compareTo(T o);
}
```

Where is Comparable used?

Collections.sort()

Arrays.sort()

SortedSet / SortedMap implementations

Useful library classes that implement Comparable

- String
- Long/Integer/Character/etc
- Date
- File

Example

```
public class Country implements Comparable<Country> {
    private String name;
    private int population;     // in millions
    // Constructor, etc …
    public int compareTo (Country other) {
        return this.population – other.population;
    }
}
```

Example (continued)

```
List<Country> countries = new ArrayList<>();
countries.add (new Country ("USA", 328));
countries.add (new Country ("Scotland", 5));
countries.add (new Country ("China", 1393));

Collections.sort (countries);
// countries now contains [Scotland, USA, China]
```

equals() and compareTo()

Documentation for `Comparable`:

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."