Java Programming 2
Lecture #11

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
28 October 2019

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

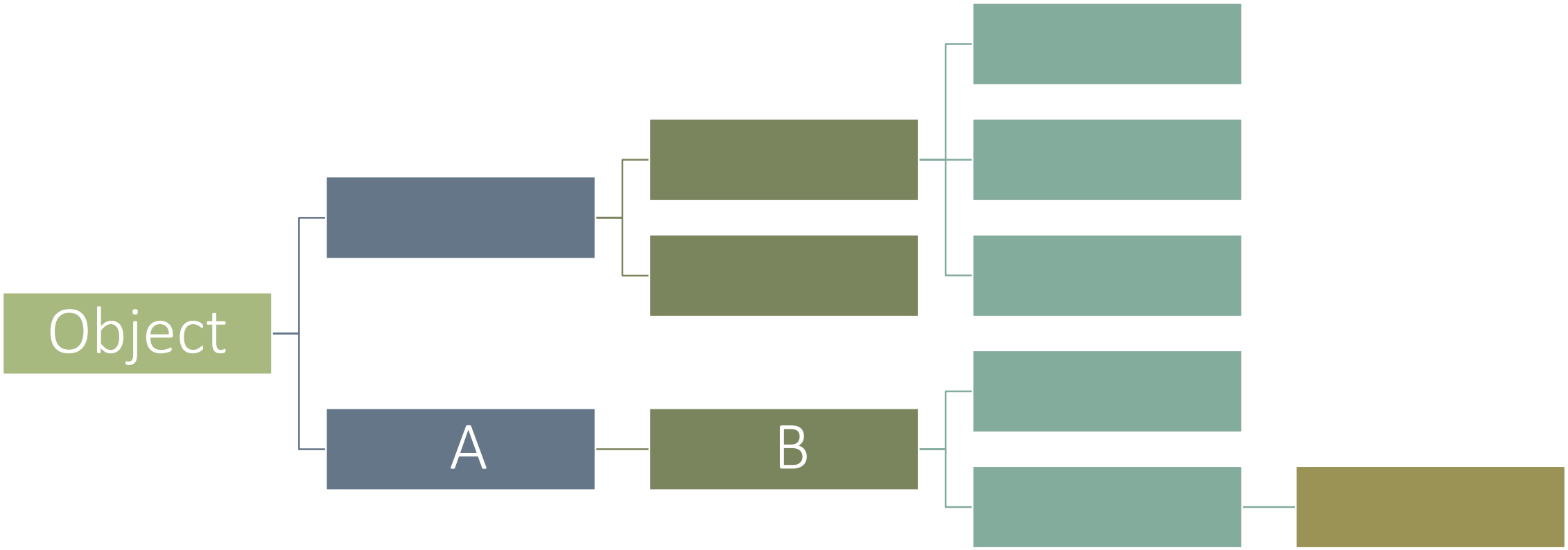
Outline

Interfaces

The `Comparable` interface

Interfaces

Inheritance in Java

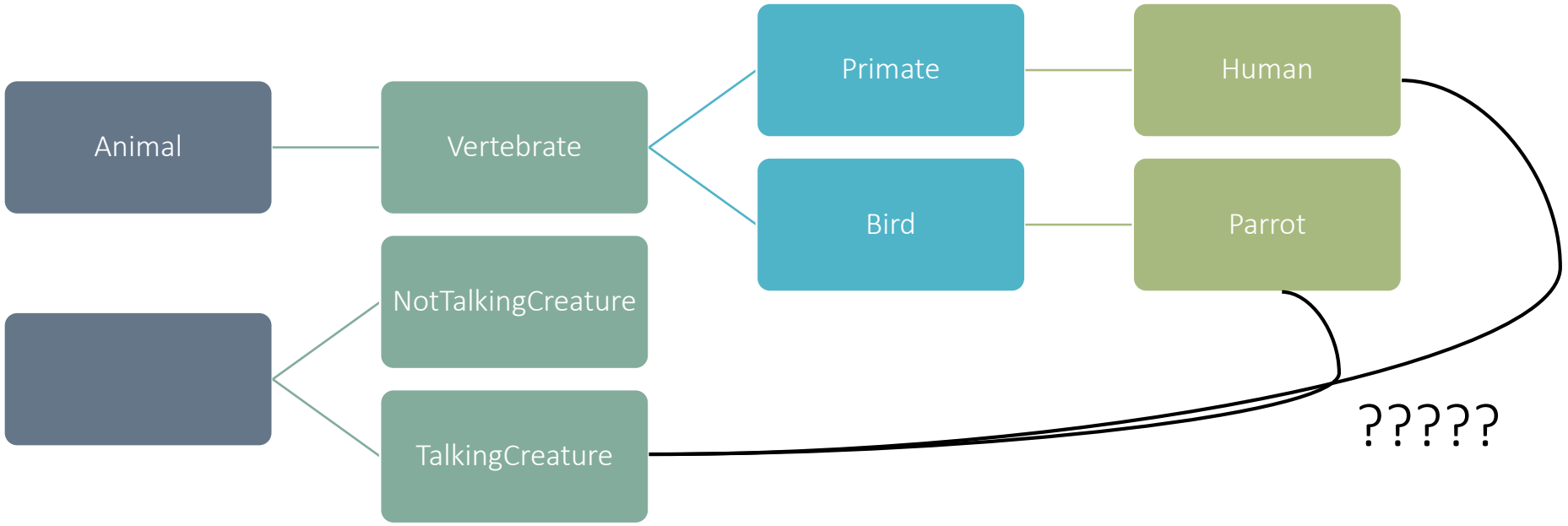


What about *multiple* inheritance?

Example:

A `Person` is both a `Primate` and a `TalkingCreature`

A `Parrot` is both a `Bird` and a `TalkingCreature`



Multiple inheritance of **state**

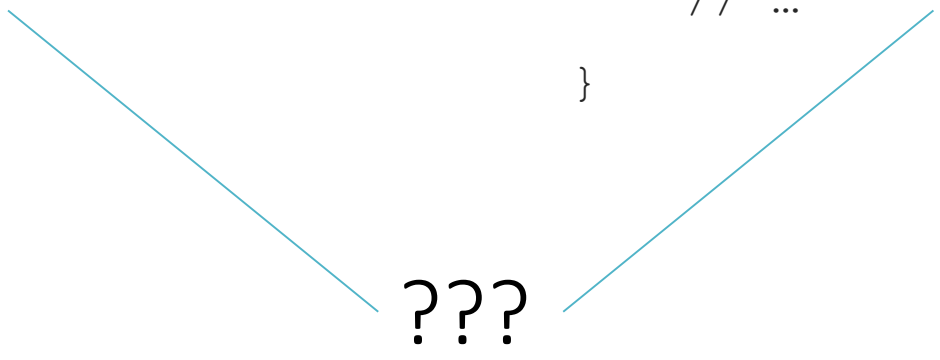
Inheriting fields from multiple classes

```
public class Class1 {
    protected int field1 = 10;
    protected String field2;

    // …
}
```

```
public class Class2 {
    protected int field1 = 5;
    protected int field2;

    // …
}
```



Multiple inheritance of **implementation**

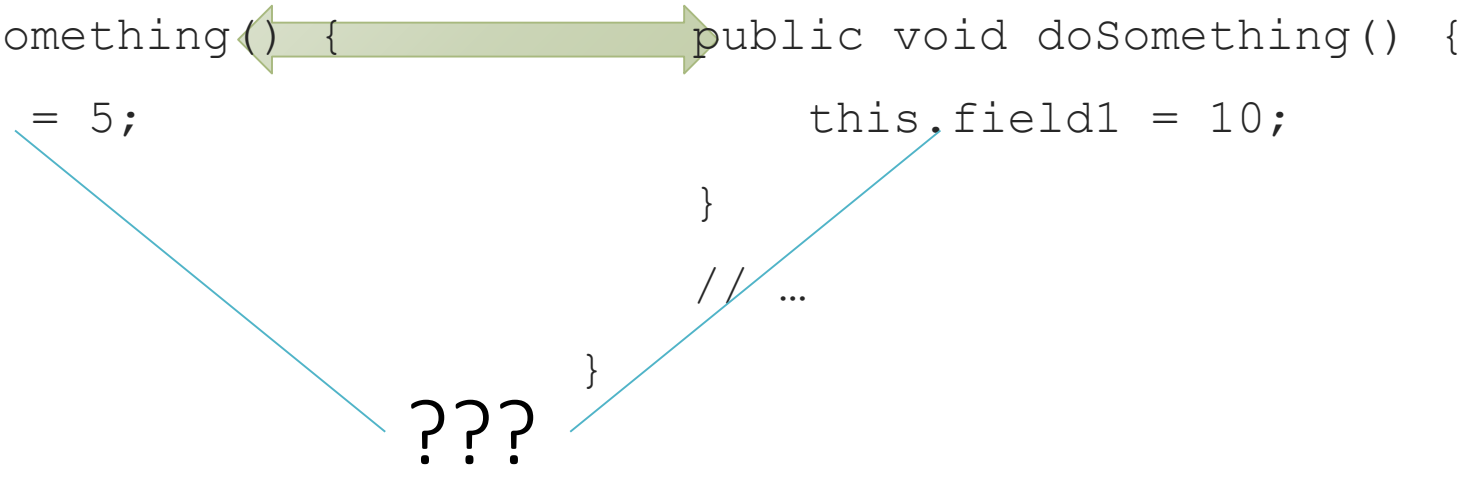
Inheriting method definitions from multiple classes

```
public class Class1 {
    protected int field1;

    public void doSomething() {
        this.field1 = 5;
    }
    // …
}
```

```
public class Class2 {
    protected int field1;

    public void doSomething() {
        this.field1 = 10;
    }
    // …
}
```



Multiple inheritance of **type**

Ability of a class to implement more than one interface (i.e., method signature only)

```
public interface Interface1 {
    public void doSomethingElse();
}
```

```
public interface Interface2 {
    public void doSomething();
}
```

```
public class MyClass implements Interface1, Interface2 {
    public void doSomething() {
        // …
    }

    public void doSomethingElse() {
        // …
    }
}
```

Interface overview

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

A class can implement any number of interfaces (including zero)

An interface specifies a public API

- Implementation is irrelevant – method signatures only

It is a **contract** that all implementing classes must honour

May also be generic – example of that coming later

Interfaces in Java

Similar to a class, but …

- Declared with `interface` keyword
- All (non-`static`) methods are implicitly `public abstract`
- All fields are implicitly `public static final`
  - *i.e., constants*
- Has no* instance-level fields or methods
  - *Eliminates issues with multiple inheritance of state and implementation*

```
public interface TalkingCreature {

    void speak(String s);

}
```

```
public interface List {
    int size();
    boolean isEmpty();
    boolean contains (Object o);
    boolean remove (Object o);
    void clear();
    // …
}
```

* Except for default methods (introduced in Java 8) … see slide #16 for details

Implementing an interface

- Use `implements` keyword
  - Conventionally, comes after `extends`
- Provide a definition for all methods declared in each interface
  - … or else declare class as `abstract`
  - All method implementations must be `public`
- Classes can implement multiple interfaces (comma-separated)

```
public class Person
    extends Primate
    implements TalkingCreature {

    public void speak (String s) {
        // …
    }

}

public class Parrot
    extends Bird
    implements TalkingCreature {

    public void speak (String s) {
        // …
    }

}
```

Interface inheritance

Interfaces can `extend` other interfaces

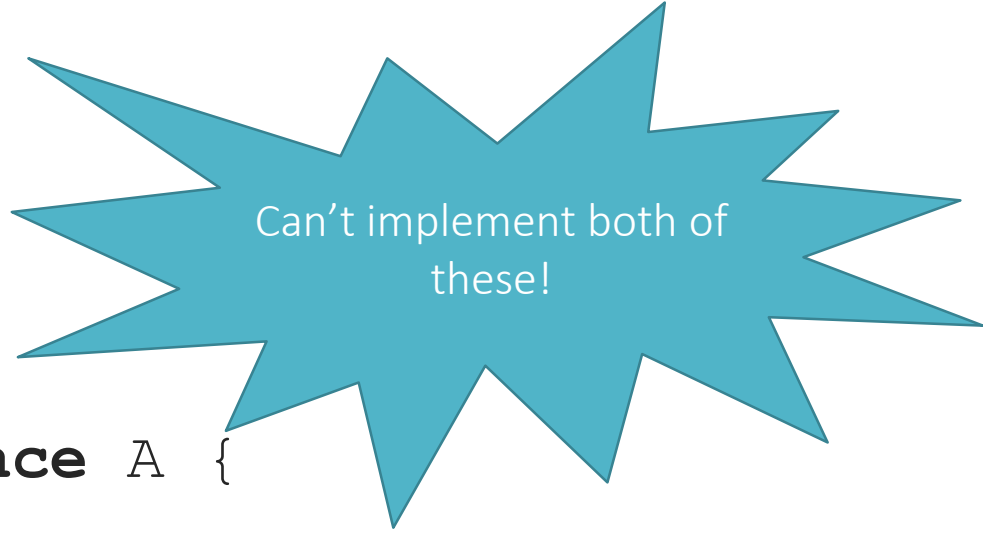
They can even extend **multiple** other interfaces!

… Because they avoid issues of multiple inheritance

Comma-separated list of parent interfaces

Interfaces cannot extend classes, and vice versa

Conflicting methods



If two interfaces have methods with the same name and parameters, but **different return types**, you cannot

- Implement them both in a class
- Inherit from both in an interface

If the two methods have **identical signatures** then it will work

- (… but possibly be confusing)

```
interface A {
    int doSomething();
}

interface B {
    String doSomething();
}
```

Using an interface as a type

You can use an interface name anywhere you use any other data type name

- Variable declarations
- Method parameters
- Generic type arguments

You cannot directly create an instance of an interface (through `new`)

```
TalkingCreature c = new Person();

public void listen (TalkingCreature c)
{
    // …
}

List<TalkingCreature> list =
    new ArrayList<>();
```

Complication: default methods

Interface methods can be declared as **default**

- Allows an implementation of the method to be provided

Implementing classes can …

- Not mention the method – default behaviour is inherited
- Redeclare the method – becomes abstract (like a normal interface method)
- Redefine the method – overriding behaviour as usual

Why was this added?

- To allow interfaces to be extended/updated without breaking all existing implementations of the interface

Interface vs. abstract class

INTERFACE

- Cannot be instantiated
- Has no constructor
- All methods are public
- Contain only constant fields
- Classes can implement **multiple** interfaces

ABSTRACT CLASS

- Cannot be instantiated
- Has a constructor
- Methods can have any access modifier
- Contain constant and “normal” fields
- Classes can have **at most one** parent class

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", 327));
countries.add (new Country ("Scotland", 5));
countries.add (new Country ("China", 1386));

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

Implementing compareTo

Useful built-in methods in many classes:

```
compare()      // static
compareTo()    // instance method, because of Comparable
```

For example, in `java.lang.Long` class:

`static int`	**`compare`**`(long x, long y)` Compares two `long` values numerically.
`int`	**`compareTo`**`(`**`Long`**` anotherLong)` Compares two `Long` objects numerically.

Another way to do comparison in Country

```
public int compareTo (Country other) {
    return Integer.compare (this.population,
           other.population);
}
// or …
public int compareTo (Country other) {
    return new Integer (this.population).
           compareTo(other.population);
}
```

More general form of comparison: `Comparator`

When to use:

If the thing you are sorting does not have a natural ordering

Or if you want to have precise control over the sort (e.g., reverse order, etc)

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Sorting countries by name with Comparator

```
public class NameComparator implements Comparator<Country> {

	public int compare(Country o1, Country o2) {
		return o1.getName().compareTo(o2.getName());
	}

}
```

Next time

File input and output