Java
Abstract and final classes, interfaces

Image: “Serious orange juicing” by Leonid Mamchenkov
https://flic.kr/p/oXdD42

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`

`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*

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

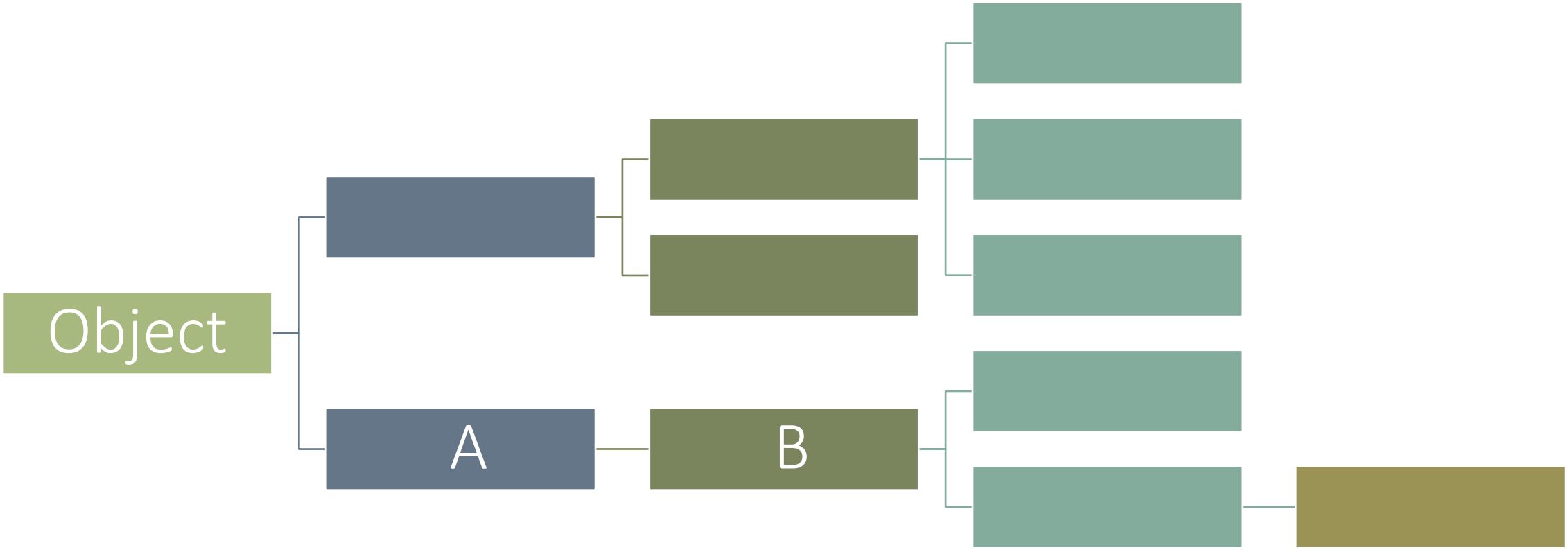
- Constructors
- Fields
- Normal (non-abstract) methods
- Static fields and methods

(Opposite of abstract)

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

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

Inheritance in Java

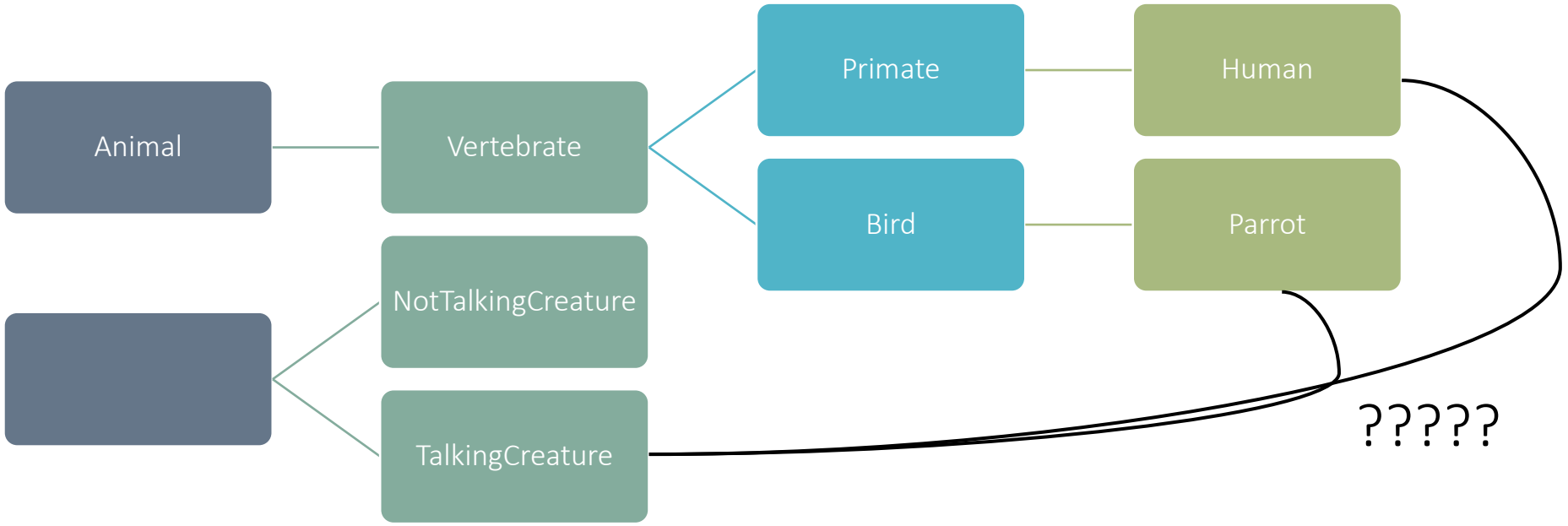


What about *multiple* inheritance?

Example:

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

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



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

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 https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html 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) {
        // …
    }

}
```

Using an interface as a type

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

- Variable declarations
- Method parameters
- etc

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

```
TalkingCreature c = new Person();

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

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