Understanding Interfaces and Polymorphism

Dan Geabunea
SENIOR SOFTWARE DEVELOPER

@romaniancoder

Overview

- Declaring interfaces
- Defining static and default methods
- Functional interfaces
- Understanding polymorphism
- Demo: Using interfaces to model airspace sectors

Interfaces

An interface is a contract that you define for the classes that implement it

Interface

A reference type that can contain method signatures, default methods, static methods, nested types and constants

Interface Restrictions

- They can not be instantiated
- An interface can not contain a constructor
- Usually, interface methods do not have a body
- Default and static methods have been introduced in recent versions of Java for compatibility reasons
- A class can implement one or more interfaces

Why Use Interfaces?

Abstraction

Hide away implementation details and only expose the public API

"Mimic" Multiple Inheritance

A class can implement multiple interfaces

Declaring and Implementing an Interface

```
interface Engine {
    void start();
    void stop();
}
```

```
public class TurboProp implements Engine {
}
```

Declaring and Implementing an Interface

```
interface Engine {
    void start();
    void stop();
}
```

```
public class TurboProp implements Engine {
    @Override
    public void start() {
        // engine start logic
    }
    @Override
    public void stop() {
        // engine stop logic
    }
}
```

Interface Access Modifiers

```
interface Engine {
    void start();
    void stop();
}
```

Interface methods are by default abstract and public

When a class implements them, they must be public

We can add the “public” access modifier to interface methods

Interface Access Modifiers

```
interface Engine {

    public void start();

    public void stop();

}
```

Interface methods are by default abstract and public

When a class implements them, they must be public

We can add the “public” access modifier to interface methods

Declaring Constants in Interfaces

```
interface Engine {

        // public, static, final by default

    int MIN_OPERATING_TEMPERATURE = -50;

    public void start();

    public void stop();

}

// Access it

int minTemp = Engine.MIN_OPERATING_TEMPERATURE;
```

In Java, interfaces can also contain constants

Fields defined in interfaces are public, static and final by default

Default and Static Methods

Evolving Interfaces

```
interface Engine {
    void start();
    void stop();
}
```

```
public class TurboProp implements Engine {
    @Override
    public void start() {}

    @Override
    public void stop() {}
}
```

Evolving Interfaces

```
interface Engine {
    void start();
    void stop();
    String healthCheck();  // new
}
```

```
public class TurboProp implements Engine {
    // Error, this class will not compile
    // because it must override healthCheck()

    @Override
    public void start() {}

    @Override
    public void stop() {}
}
```

Changing interfaces can break classes that implement them

Default method

A method defined in an interface that has an implementation. They allow interfaces to evolve but don’t force changes to classes that implement those interfaces

Default Methods

```
interface Engine {
    void start();
    void stop();
    default String healthCheck() {
        // logic goes here
            return “OK”;
    }
}
```

```
public class TurboProp implements Engine {
    @Override
    public void start() {}

    @Override
    public void stop() {}
}

Engine e = new TurboProp();
e.healthCheck();
```

Properties of Default Methods

Declared with the “default” keyword at the beginning of the method signature

Must provide an implementation

They are implicitly public

Can be overridden by classes that implement the interface

Override Default Methods

```
interface Engine {
    void start();
    void stop();
    default String healthCheck() {
        // logic goes here
            return “OK”;
    }
}
```

```
public class TurboProp implements Engine {
    @Override
    public void start() {}

    @Override
    public void stop() {}

    @Override
    public String healthCheck() {…}
}
```

Static Methods in Interfaces

A method that “belongs” to the interface. You can declare a full-fledged method as static and use the interface name to call it

Static Interface Methods

Are Related to the Interface

```
interface Engine {
    int MIN_TEMP = -50;
    void start();
    void stop();

    static boolean canStart(int outsideTemp) {
        return outsideTemp > MIN_TEMP;
    }
}
```

```
// Use interface name, followed by static method
boolean canStart = Engine.canStart(-30);
```

Static Interface Methods

No Relationship with the Classes That Implement the Interface

```
interface Engine {
    int MIN_TEMP = -50;
    void start();
    void stop();

    static boolean canStart(int outsideTemp) {
        return outsideTemp > MIN_TEMP;
    }
}
```

```
// A static interface method is linked to the
// interface, not to classes that implement it
boolean canStart = TurboProp.canStart( 30);
```

Defining a static method in an interface is the same as defining it in a class

If you want a placeholder to store static methods, then you really don't need a concrete class for this

Functional Interfaces

Functional Interface

An interface that contains a single abstract method

It is an interface that exposes one single responsibility

Functional Interface

```
@FunctionalInterface
public interface EventHandler {
    void handle();   // a singe method
}
```

Starting with Java 8, we can use a lambda expression to implement a functional interface

Implementing Functional Interfaces with Lambda Expressions

```
@FunctionalInterface
public interface EventHandler {
    void handle();
}
```

```
EventHandler onStart = () -> {
    System.out.println("Program started");
};

onStart.handle();
```

Widely Used Functional Interfaces

Predicate<T>

Function<T>

Runnable

Comparable<T>

```
List<String> manufacturers = List.of("Boeing", "Airbus", "Embraer");
// Print manufacturers that start with A


Predicate<String> startWithA = (String val) -> {
    return val.startsWith("A");
};


manufacturers.stream()
            .filter(startWithA)
            .forEach(System.out::println);
```

Properties of Functional Interfaces

Can have a single abstract method, but multiple default or static methods

We can use @FunctionalInterface to make sure the interface does not have more than one abstract method

We can implement functional interfaces with lambdas, no need for creating concrete classes

Polymorphism

Polymorphism

The ability of an abstraction to take many forms at runtime

Method Overriding

Inheritance

Override behavior from the superclass, to create more specialized implementations

Interfaces

Classes that implement the interface must override the intent defined in the interface

```
// Depending on model, each radar uses different formats
public interface RadarInterface {
    List<RadarTarget> readData();
}
```

Implementing Interfaces

Many Forms for Same Intent

```
public class BinaryRadar implements RadarInterface{
    @Override
    public List<RadarTarget> readData() {
        // read binary data and extract targets
    }
}
```

```
public class XmlRadar implements RadarInterface {
    @Override
    public List<RadarTarget> readData() {
        // parse xml and extract radar targets
    }
}
```

```
// Combine data from multiple radars
List<RadarInterface> radars = List.of(
        new XmlRadar(),
        new BinaryRadar());
List<RadarTarget> combinedData = new ArrayList<>();
radars.forEach(r -> combinedData.addAll(r.readData()));
```

Runtime Behavior

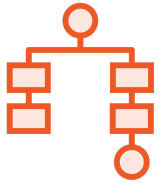
We are using a common reference variable type

At runtime, the behavior that gets triggered is determined by the actual object type (the objects we create)

Benefits of Polymorphism

Change behavior of an application at runtime even without recompiling your code

Reduces coupling because we can depend on abstractions, not concrete types



We can use a single variable type to store many types

Interfaces vs. Abstract Classes

What should I use?

Interfaces vs. Abstract Classes

Use abstract classes

When you want to offer some base functionality to subclasses

Provide a template for future classes

Create abstract members that are not public

Use interfaces

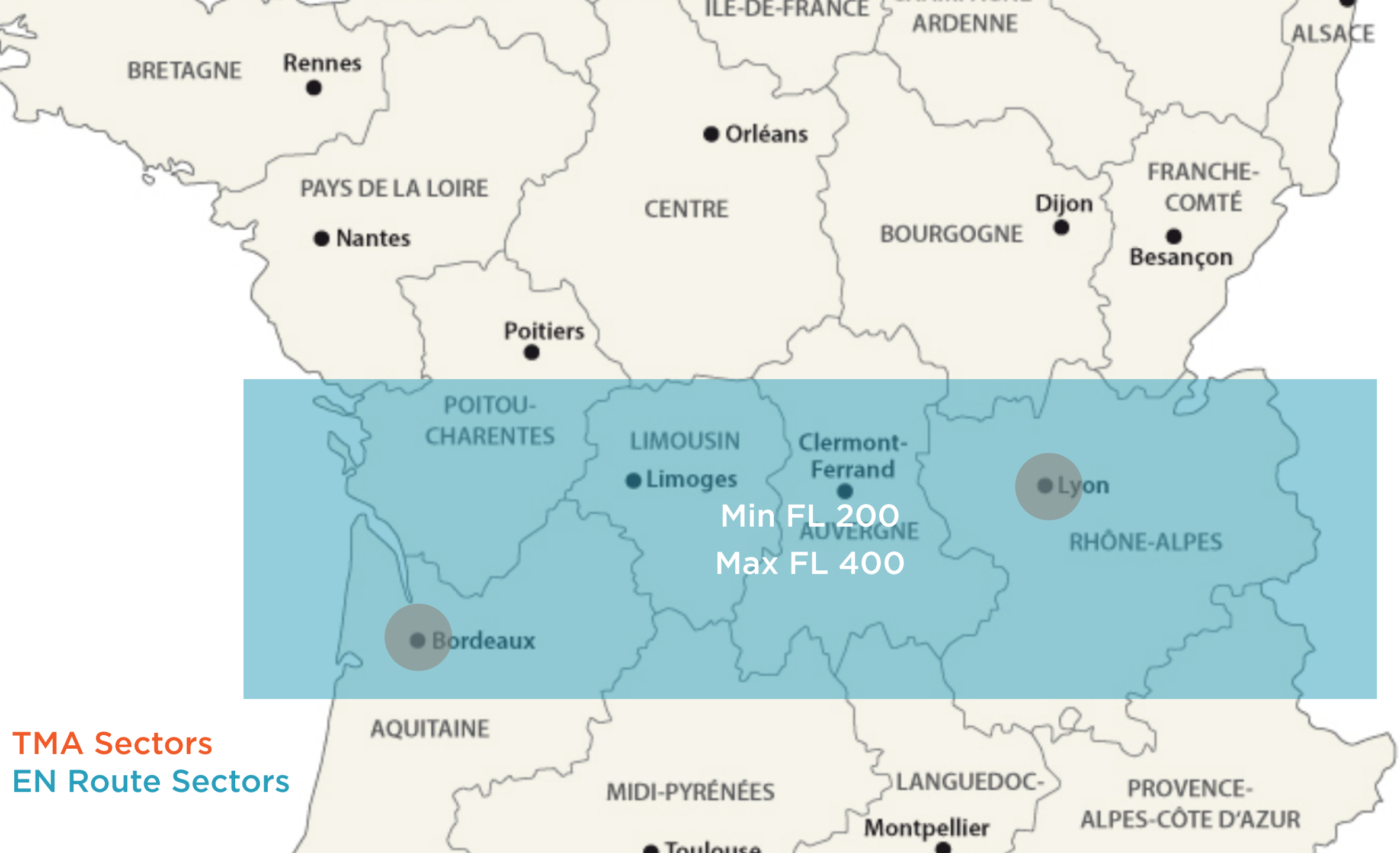
Highest level of abstraction

Highest level of loose coupling

Implement more interfaces

Demo

Demo: Using interfaces and polymorphism to implement airspace sectors



TMA Sectors

EN Route Sectors

Summary

Interfaces are a reference type that usually holds only abstract methods

We can think about interfaces like a public contract that each class must respect

Default methods allow interfaces to evolve without breaking existing code

Functional interfaces are interfaces with a single abstract method. They can be implemented using lambda expressions

Interfaces are great for implementing polymorphism. The intent they define can take many forms at runtime based on the objects that were created from them

Up Next:
Defining Enumerations and Nested Classes