Defining Enumerations and Nested Classes

Dan Geabunea
SENIOR SOFTWARE DEVELOPER

@romaniancoder

Overview

Defining and using enum types

Enhancing enum types with properties and methods

Demo: Using enums to model an aircraft wake turbulence category

Inner classes

Local classes

Anonymous classes

Lambda Expressions

Demo: Using inner classes to validate complex types

Enum Types

Enum Type

A data type that enables a variable to hold only certain predefined values. They should be used every time you need to represent a fixed set of constant values.

Define Enum

```
public enum FlightRules {

    INSTRUMENT_FLIGHT_RULES,

    VISUAL_FLIGHT_RULES,

    SPECIAL_FLIGHT_RULES

}


// Notice the naming conventions

// Use uppercase because we are defining a set of

// constants
```

Use Enum

```
int calculateMinSeparation(FlightRules fr) {
    switch (fr) {
      case VISUAL_FLIGHT_RULES:
        return 20;  // Nm
      case INSTRUMENT_FLIGHT_RULES:
        return 10;  // Nm
      case SPECIAL_FLIGHT_RULES:
        return 15;  // Nm
      default:
        return -1;
    }
  }
```

Enums in Java are more powerful than enums in other languages

Enum Type

The enum declaration defines a special class called enum type

An enum type can have a body which can include fields and methods

The compiler also adds some special static methods like values() or valueOf()

An enum type can have properties assigned to each constant value

An enum type can have a constructor, which can be used to assign properties to enum constants

```
public enum FlightRules {
      // Constants must be defined first; assign separation as property on constants
   INSTRUMENT_FLIGHT_RULES(10),
   VISUAL_FLIGHT_RULES(20),
   SPECIAL_FLIGHT_RULES(15);

   private int minSeparation;        // Property assigned to constant

      // Constructor must be private or package private
   FlightRules(int minSeparation) { this.minSeparation = minSeparation; }

   public int getMinSeparation() { return minSeparation; }
}
```

Use Enum Methods

```
public int calculate(FlightRules fr) {
    return fr.getMinSeparation();
}

calculate(FlightRules.INSTRUMENT_FLIGHT_RULES);          // Output: 20
```

Enum Type Restrictions

The constants must be declared at the top of the enum body; Everything else will be declared after them

The constructor of the enum type must be private or package private; It automatically creates the constants defined in the enum body

You can not call the enum type constructor directly

Demo

Demo: Using enums to model an aircraft wake turbulence category

Inner Classes

Nested Class

In Java we can define a class within another class; Such a class is called a nested class

Nested Classes

Static Nested Classes

Inner Classes

Inner Class

An inner class is a class associated with an instance of its outer class

Inner Class Characteristics

✓ It has direct access to the outer class object's fields and methods

✓ Because it is associated with an instance of the enclosing class, it can not contain any static members

✓ To instantiate an inner class, you must first instantiate the outer class. Then you can create an inner class object using the outer class object

Reduced Vertical Separation Minima

To increase the number of aircraft that can fly in an airspace by reducing the minimum required vertical distancing from 2000 ft to 1000 ft.

Conditions: The aircraft altitude must be between FL 290 and FL 410 and the aircraft must be RVSM capable

```java
public class Aircraft {

    private final int altitudeFl;

    private final boolean isRvsmCapable;


    public int getSeparationFeet() {

        // Logic goes in here

        // Check altitude and RVSM capability to determine separation

    }

}
```

Private Inner Class

```
public class Aircraft {
    private final int altitudeFl;
    private final boolean isRvsmCapable;

    private class VerticalSeparation {
        private int separationInFeet;
        VerticalSeparation() {
            if (altitudeFl >= 290 && altitudeFl <= 410 && isRvsmCapable) { separationInFeet = 1000;}
            else {  separationInFeet = 2000; }
        }
        public int getSeparationInFeet() { return separationInFeet; }
    }

    public int getSeparationFeet() { VerticalSeparation vsep = new VerticalSeparation(); return vsep.getSeparationInFeet(); }
}
```

Private Inner Class

```
public class Aircraft {
    private final int altitudeFl;
    private final boolean isRvsmCapable;

    public class VerticalSeparation {
        private int separationInFeet;
        VerticalSeparation() {
            if (altitudeFl >= 290 && altitudeFl <= 410 && isRvsmCapable) { separationInFeet = 1000;}
            else {  separationInFeet = 2000; }
        }
        public int getSeparationInFeet() { return separationInFeet; }
    }

    public int getSeparationFeet() { VerticalSeparation vsep = new VerticalSeparation(); return vsep.getSeparationInFeet(); }
}
```

```
// Define an instance of the outer class

Aircraft a = new Aircraft(300, true);


// Using the outer class instance create a new inner class instance

Aircraft.VerticalSeparation vsep = a.new VerticalSeparation();


System.out.println(vsep.getSeparationInFeet());
```

Instantiating Inner Classes

An instance of the inner class can only exist within an instance of the outer class

Special Types of Inner Classes

Local classes

Anonymous classes

Local Classes

Local Class

A class that is defined within a block of code, usually within a method

Where Can You Define a Local Class

In a method

In a for loop

In an if clause

```
public class Aircraft {
    private final int altitudeFl;
    private final boolean isRvsmCapable;

    public int getSeparationFeet() {
        class VerticalSeparation {                          // No access modifier
            private int separationInFeet;
            VerticalSeparation() {
                if (altitudeFl >= 290 && altitudeFl <= 410 && isRvsmCapable) { separationInFeet = 1000; }
                else {  separationInFeet = 2000; }
            }
            public int getSeparationInFeet() { return separationInFeet; }
        }
        VerticalSeparation vsep = new VerticalSeparation(); return vsep.getSeparationInFeet();     // Must be instantiated in same block
    }
}
```

Access Members of Outer Class

- ✓ A local class can access all the members of its enclosing class

- ✓ In addition to that a local class can access the local variables defined in the same scope; But these variables need to be final or effectively final

- ✓ Local classes can access the method parameters if they are defined within a method

Accessing Local Fields

A Local Class Can Access Final or Effectively Final Local Variables

```java
public class Conversions {
    public int fromFeetToFL() {
        final int valueInFeet = 100;        // Final local variable

        class FeetToFL {
            public int get() { return valueInFeet / 100; }    // Can access final local variable
        }

        FeetToFL convertor = new FeetToFL();
        return convertor.get();
    }
}
```

Accessing Local Fields

A Local Class Can Access Final or Effectively Final Local Variables

```
public class Conversions {

    public int fromFeetToFL() {

        int valueInFeet = 100;     // Effectively final


        class FeetToFL {

            public int get() { return valueInFeet / 100; }   // Can access final local variable

        }


        FeetToFL convertor = new FeetToFL();

        return convertor.get();

    }

}
```

Accessing Local Fields

```
public class Conversions {
    public int fromFeetToFL() {
        int valueInFeet = 100;
              valueInFeet = 200;    // Not final or effectively final anymore

        class FeetToFL {
            ~~public int get() { return valueInFeet / 100; }~~  // Error
        }

        FeetToFL convertor = new FeetToFL();
        return convertor.get();
    }
}
```

Local Class Restrictions

- They can not contain any static members, except constants (final static fields of primitive types or String)

- You can not declare interfaces in a block, just classes

- They can not be instantiated from outside the block they were defined in

- They do not have access modifiers since they are defined within a block and used within the same block

Anonymous Classes

Anonymous Class

Simplified local class; A great way to declare and instantiate a class at the same time

Anonymous Classes Are Expressions

```
// The interface can have as many members as possible

public interface UnitConvertor {

    int convert();

}
```

```
public void someMethod() {

    int feet = 2000;          // Final or effectively final

    UnitConvertor feetToFl = new UnitConvertor() {

        @Override

        public int convert() {

            return feet / 100;

        }

    };

    System.out.println(feetToFl.convert());

}
```

```
public void someMethod() {
    int feet = 2000;


    UnitConvertor feetToFl = new UnitConvertor() {
      @Override
      public int convert() { return feet / 100; }
    };


    UnitConvertor feetToMeters = new UnitConvertor() {
      @Override
      public int convert() { return (int) (feet * 0.3048); }
    };
  }
```

```
UnitConvertor feetToFI = new UnitConvertor() {
    @Override
    public int convert() {
        return feet / 100;
    }
};
```

Anonymous Class Expression

The new operator

The name of an interface/base class that needs to be implemented or extended

Parentheses that can contain arguments to a constructor

A body in which we define the class

An anonymous class expression is almost like invoking a constructor, except you need to define a class in a block of code

Access Members of Outer Class

✓ An anonymous class can access all the instance members of its enclosing class

✓ In addition to that an anonymous class can access the local variables defined in the same scope; But these variables need to be final or effectively final

✓ Anonymous classes can access the method parameters if they are defined within a method

Anonymous Class Restrictions

They can not contain any static members, except constants (final static fields of primitive types or String)

You can not declare constructors in them

```java
public void someMethod() {
    int feet = 2000;

    UnitConvertor feetToFl = new UnitConvertor() {
      @Override
      public int convert() { log(); return feet / 100; }

      // You can add extra methods
      private void log(){
        System.out.println("Converting " + feet + " to FL");
      }
    };
}
```

What You Can Declare in an Anonymous Class

Methods

Fields

Local classes

Instance initializers

If you want to define a class with only one method, then even anonymous classes are a bit too complicated

Lambda Expressions

Lambda Expression

A way to represent a functional interface using an expression; It is treated as a function by the compiler

Lambda Expression Vs. Anonymous Class

Filtering a Collection of Objects

```
public class Aircraft {

   private final String callSign;

   private final int altitudeFl;

   private final boolean isRvsmCapable;

   // Getters and setters …

}
```

```
List<Aircraft> aircraft = List.of(

            new Aircraft("OS731", 100, true),

            new Aircraft("ROT123", 120, true),

            new Aircraft("BA087", 140, false),

            new Aircraft("AF567", 250, true),

            new Aircraft("LUF676", 360, false)

      );

print(aircraft);    // print specific aircraft
```

Define Functional Interface

```
@FunctionalInterface

public interface AircraftFilter {

    boolean check(Aircraft a);

}
```

Using an Anonymous Class

```java
private void print(List<Aircraft> aircraft) {
      // Anonymous class => lots of boilerplate; What if we need to change the filters or provide them as a method param?
   AircraftFilter lowAltitudeFilter = new AircraftFilter() {
     @Override
     public boolean check(Aircraft a) {
       return a.getAltitudeFl() < 150;
     }
   };

   aircraft.forEach(a -> {
     if (lowAltitudeFilter.check(a)) { System.out.println(a.getCallsign()); }
   });
 }
```

Using a Lambda Expression

```
private void print(List<Aircraft> aircraft, AircraftFilter filter) {
    aircraft.forEach(a -> {
      if (filter.check(a)) {
        System.out.println(a.getCallsign());
      }
    });
}
```

```
// Provide filter implementation using lambdas
print(aircraft, (a -> a.getAltitudeFl() < 150));
print(aircraft, (a -> a.getAltitudeFl() > 290 && a.getAltitudeFl() < 410 && a.isRvsmCapable()));
print(aircraft, (a -> a.getAltitudeFl() < 150));

print(aircraft, (a -> {

   return a.getAltitudeFl() < 150;

}));
```

Syntax of a Lambda Expression

A comma separated list of input parameters

The arrow token ->

A body which can be a single expression or a method block

Lambda expressions can capture final or effectively final local variables of the enclosing scope

Using Lambda Expressions in Java Code

Jose Paumard

Demo

Demo: Using inner classes to validate complex types

Summary

Enum type is great for storing fixed sets of constants

Enum type is powerful in Java; You can add methods, fields and constructors

Java inner classes can access the instance members of its enclosing class while still being a reusable class itself

Local classes are defined within a block of code

Anonymous classes allow us to declare and instantiate a class at the same time based on an interface or superclass

Lambda expressions implement a single unit of behavior that can be passed to other code

Course Recap

CONGRATS!

Course Recap

- Declaring and using classes
- Adding state and behavior
- Deep dive into static fields, methods and classes
- Abstraction, Encapsulation, Inheritance and Polymorphism
- Creating code contracts with interfaces
- Using enum types and nested classes

You have all the OOP skills needed to take the **Java SE 11 Developer Certification 1z0-819** and to create more robust projects by leveraging OOP effectively

Follow the rest of the path to sharpen all the skills needed for the exam

Source Code

https://github.com/dangeabunea/**pluralsight-java11-object-oriented-approach**

Dan Geabunea

Let’s get in touch

- @romaniancoder
- https://ro.linkedin.com/in/dangeabunea

My other Java Pluralsight courses

- SOLID Software Design Principles in Java
- Spring Framework: Spring Data MongoDB