Lecture 16 — Enums and Autoboxing

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 12 of the textbook
- Enums – types for enumerations
- Autoboxing/unboxing of primitive types

Representing Days

- Consider writing a method that determines if a day is a weekday or weekend
- How do we represent the day of the week?
- We could do something like: `boolean isWeekend(int dayOfWeek)`
- Let 1="Monday", 2="Tuesday", 3="Wednesday", 4="Thursday", 5="Friday", 6="Saturday", 7="Sunday"
- Now we need to check if the input is a valid number and potentially throw an exception if it isn't
- Also, we need to remember this mapping every time we write or modify code
- 1="Sunday", 2="Monday", is equally as valid

Representing Transport

- Consider writing a program to model a public transport network
- Methods of transport include: bus, train, ferry, tram
- We might implement a method to get the typical speed of a transport type
- We could represent the types as a String:
  `int typicalSpeed(String methodOfTransport)`
- We end up with similar issues as for `int`: we now need to deal with invalid or null Strings

Introducing Enum

- Java has a way of enforcing a data type has a limited range of values
- This is using an *Enum*
- It is designed to deal with cases like those given previously
- Enum is short for enumeration
- The set of values we care about is enumerated: Monday, Tuesday, Wednesday ...
- They are effective when we want to create a new type with a limited range of well defined values

enum

```
enum DayOfWeek {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}

enum Transport {
    BUS, TRAIN, FERRY, TRAM
}
```

- It is convention to use capitals (e.g., BUS)
- But any style is allowed

enum

```
enum DayOfWeek {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}

enum Transport {
    BUS, TRAIN, FERRY, TRAM
}
```

- Defines a new type we can use
- Specific enum values are accessed like this: `DayOfWeek today = DayOfWeek.Monday;`

isWeekend()

```
public static boolean isWeekend(DayOfWeek day) {
    return day == DayOfWeek.Saturday || day == DayOfWeek.Sunday;
}
```

- Since `DayOfWeek` is now a type, we can use it as a parameter
- Enum values can be compared for equality using ==
- It is now impossible to pass incorrect values as arguments!

typicalSpeed()

```
public static int typicalSpeed(Transport transport) {
    switch (transport) {
        case BUS:
            return 50;
        case TRAIN:
            return 100;
        case FERRY:
            return 20;
        case TRAM:
            return 30;
        default:
            return 0;
    }
}
```

- Enums work with `switch` statements too

Example Usage

```
public static void main(String[] args) {
    DayOfWeek day = DayOfWeek.Monday;
    System.out.println(isWeekend(day));
    System.out.println(isWeekend(DayOfWeek.Saturday));

    System.out.println(typicalSpeed(Transport.BUS));
    System.out.println(typicalSpeed(Transport.TRAIN));
}
```

- Here is an example usage
- In Java, `enum` is a special kind of class
- However, we do not use `new`. Instead, we access the finite range of values via `EnumName.MEMBER`

Enums are Classes

- Enums can be `public`, `protected`, `private`, or default like any class
- However, they cannot extend or be extended through inheritance
- All enums implicitly extend the built-in `Enum` class
- This means they come with several useful methods, which we will see
- `values()`, `ordinal()`, `valueOf()`
- You can also add your own methods

Example Usage

```
public class EnumMethods {
    public static void main(String[] args) {
        // Go through all the days of the week
        for (DayOfWeek day : DayOfWeek.values()) {
            System.out.println(day);
        }
        int dayNumber = DayOfWeek.Tuesday.ordinal();
        String dayName = DayOfWeek.Tuesday.name();
        DayOfWeek day = DayOfWeek.valueOf("Tuesday");
        System.out.println(dayNumber);
        System.out.println(dayName);
        System.out.println(day);
    }
}
```

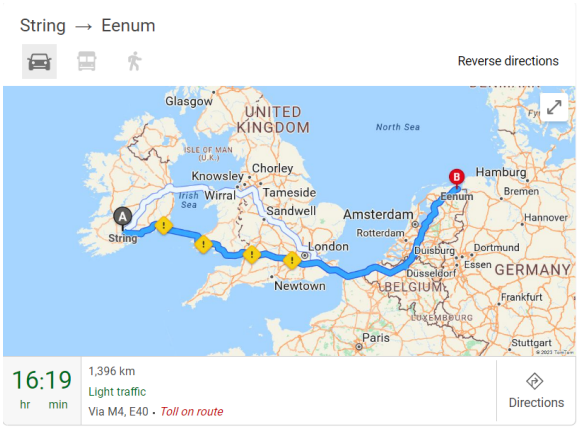
Enums Fields and Constructors

- As mentioned, it is possible to add your own methods to enum classes
- It is also possible to add fields and constructors
- Note that constructors work a little different to usual
- This is because we do not construct enums directly (e.g., using `new`)

Enum Constructor

```
enum Transport {
    BUS(50), TRAIN(100), FERRY(20), TRAM(30);

    private final int typicalSpeed;

    Transport(int typicalSpeed) {
        this.typicalSpeed = typicalSpeed;
    }

    public int getTypicalSpeed() {
        return typicalSpeed;
    }
}

public class EnumConstructor {
    public static void main(String[] args) {
        System.out.println(Transport.BUS.getTypicalSpeed());
        System.out.println(Transport.TRAIN.getTypicalSpeed());
    }
}
```

Mid-lecture Break



Autoboxing and Unboxing

- Java provides built-in object types for each primitive type
- Integer, Double, Character, Short, Byte, Float, Boolean, etc.
- Autoboxing: automatic conversion from primitive to object type
- Unboxing: automatic conversion from object to primitive type
- Useful for data structures or methods that only accept object types

Autobox List

```
import java.util.*;
public class AutoboxList {
    public static void main(String[] args) {
        // Note ArrayList implements List
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(3);
        int sum = 0;
        for (int i : list) {
            sum += i;
        }
        Integer four = 4;
        sum += four;
        System.out.println(sum);
    }
}
```

Autoboxing and Unboxing Methods

- Autoboxing applies to method parameters
- It also applies to method return types
- In general, anywhere a primitive would usually be expected, a value gets autoboxed
- Anywhere an object is expected, it is unboxed

Autobox Method

```
public class AutoboxMethod {
    public static int sum(int a, int b) {
        return a + b;
    }

    public static Double max(Double a, Double b) {
        return a > b ? a : b;
    }

    public static void main(String[] args) {
        Integer a = 1, b = 2;
        int sum = sum(a, b);
        System.out.println(sum);
        Double max = max(1.0, 2.0);
        System.out.println(max);
    }
}
```

Multiple References

```
public class MultipleRefs {
    public static void main(String[] args) {
        Integer a = 1;
        Integer b = a;
        a += 2;
        System.out.println(a);
        System.out.println(b);
    }
}
```

- Note that autoboxing/unboxing cannot be used to share multiple references to a primitive type
- To achieve this, you would need to make your own wrapper class

Autoboxing and Performance

- Autoboxing/unboxing comes with a performance hit
- It involves creating a new object
- Using primitive types is much faster and more space efficient
- In general, always use the primitive type unless you need autoboxing
- When would you need autoboxing?
- As we have seen, many API data structures (such as ArrayList) expect an object
- Why can't they just handle primitive types too?
- We will see the answer in a later lecture on *generics*, which are how these data structures are implemented

Integral Classes

- Classes such as `Boolean`, `Long`, `Double`, come with useful methods
- Integral types such as `Long` and `Integer` come with methods specific to those types
- Converting Strings to integers with specific bases: `parseInt()`
- Note that this can throw a checked `NumberFormatException`
- Dealing with the binary representation of integers: `highestOneBit()`, `toBinaryString()`
- Numeric limits: `MAX_VALUE`, `MIN_VALUE`
- Many more
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Integer.html`

Floating Point Classes

- Floating point types (`Float` and `Double`) have similar methods
- Converting Strings to doubles `parseDouble()`
- Note that this can throw a checked `NumberFormatException`
- Special constants: `NaN`, `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`
- Numeric limits: `MAX_VALUE`, `MIN_VALUE`
- Many more
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Double.html`

Character Class

- The `Character` class has many useful methods, some of which we have already seen before
- The categories of character:
  `isDigit()`, `isLetter()`, `isUpperCase()`, `isLowerCase()`, `isWhiteSpace()`
- Conversions: `toUpperCase()`, `toLowerCase()`
- Many more
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Character.html`