EECS 2030 Advanced Object-Oriented Programming

S2023, Section A

Enumerated Types

So far…

- How to store and organize data?
  - Primitive types
  - Reference types
    - Composition, Aggregation
    - Inheritance
- Do we need anything else?

Motivation

consider the following groups:

- days of the week
- months of the year
- suits in a standard deck of playing cards
- ranks in a standard deck of playing cards
- Canadian coins
- Canadian provinces and territories
- planets of the solar system
- arithmetic operations

what do these groups have in common?

Enumerated types

an *enumerated* type (or *enum* type or *enum*) is a type whose values consist of a fixed set of constants

- Sunday, Monday, ..., Saturday
- January, February, ..., December
- clubs, diamonds, hearts, spades
- 2, 3, ..., ace
- nickel, dime, ..., toonie
- and so on

Old Style Enums

older Java code and C code used int constants to represent enumerated types

```
public static final int SUNDAY = 0;
public static final int MONDAY = 1;
public static final int TUESDAY = 2;
// and so on

public static final int JANUARY = 0;
public static final int FEBRUARY = 1;
public static final int MARCH = 2;
// and so on
```

Old Style Enums **problems**

No type safety

- any **method with an int parameter** will accept a day or month
- there is **no way to restrict** a parameter to be only a day or a month
- you **can perform arithmetic** with days and months
- you **can compare days *with* months** (for equality, inequality, less than, greater than)

Old Style Enums problems

another problem with using ints to represent enums is that there is no easy way to translate the int value to a string

- There is no **`toString`** method for ints
- Or for any *primitive* types

Old Style Enums

for enumerations such as months or days it is also common to use strings for the constants instead of ints

```
public static final String SUNDAY = "SUNDAY";
public static final String MONDAY = "MONDAY";
public static final String TUESDAY = "TUESDAY";
// and so on

public static final String JANUARY = "JANUARY";
public static final String FEBRUARY = "FEBRUARY";
public static final String MARCH = "MARCH";
// and so on
```

Old Style Enums problems

Using strings for the constants isn't much better than using ints

- all of your enumerations are now strings so there is **still no real type safety**
- it is possible to use **`"SUNDAY"`** for a **`Month`**
- comparing strings for equality is slower than comparing ints or addresses

Java enums

Java 1.5 (2004, “J2SE 5.0”) added an enum type

- true Java **classes** that implicitly inherit from the superclass **`java.lang.Enum`**
- enums can **have fields and methods** and can implement interfaces
- enums can have **constructors** but the constructors are not accessible outside of the enum (the constructors are implicitly **`private`**)

Java enums

an example of a simple enum:

```java
public enum Day {
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY;
}
```

Java enums

an example of using the Day enum:

```
Day d = Day.THURSDAY;
System.out.println(d);
```

prints:

```
THURSDAY
```

notice that enums provide a compiler generated toString method

- you can override toString if you wish

Java enums

an enum exports exactly one instance for each enumeration constant via a *public static final* field

because there are no accessible constructors there can be no instances except the ones exported by the enum

Type safety

enums are *types*; therefore you get all of the benefits of **compile-time** type safety

suppose you have a second enum type:

```
public enum Month {
  JANUARY, FEBRUARY,
  MARCH, APRIL,
  MAY, JUNE,
  JULY, AUGUST,
  SEPTEMBER, OCTOBER,
  NOVEMBER, DECEMBER;
}
```

Enums implement Comparable

enums automatically implement the **`Comparable`** interface

the natural ordering is the order in which the constants are defined

- unfortunately, there is no way for the implementer to override the behavior of **`compareTo`** in an enum

Enums implement Comparable

example using Month:

```
Day d1 = Day.MONDAY;
Day d2 = Day.THURSDAY;
Day d3 = Day.SATURDAY;
System.out.println(d1.compareTo(d2));
System.out.println(d3.compareTo(d1));
System.out.println(d2.compareTo(d2));
```

prints:

```
-3
5
0
```

Enums can have fields

- enums can have fields, and the fields can be initialized via a constructor
  - constructors for enums are always private!
- enums are supposed to be constants

Enums can have fields

```
public enum Month {
  JANUARY(31),
  FEBRUARY(28),
  MARCH(31),
  APRIL(30),
  MAY(31),
  JUNE(30),
  JULY(31),
  AUGUST(31),
  SEPTEMBER(30),
  OCTOBER(31),
  NOVEMBER(30),
  DECEMBER(31);

  private final int days;
```

Enums can have fields

```java
Month(int days) {
   this.days = days;
 }

 public int days(int year) {
   if (this != FEBRUARY) {
     return this.days;
   }
   if (year % 400 == 0 ||
          (year % 4 == 0 && year % 100 != 0)) {
     return this.days + 1;
   }
   return this.days;
 }
```

The values method

every enum has a compiler-generated public static method called **`values`**

`values` returns an array of the enumeration constants in the order that they were declared

More info: Java Language Specification

The values method

example using **Month**:

```
System.out.println(Arrays.toString(Month.values()));
```

prints (all on one line):

```
[JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER]
```

The values method

another example using **Month**:

```
System.out.println(Arrays.toString(Month.values()));

for (Month m : Month.values()) {
    System.out.println(m + " " + m.days(2022));
}
```

The values method

prints:

```
JANUARY 31
FEBRUARY 28
MARCH 31
APRIL 30
MAY 31
JUNE 30
JULY 31
AUGUST 31
SEPTEMBER 30
OCTOBER 31
NOVEMBER 30
DECEMBER 31
```

The valueOf method

every enum has another compiler generated public static method with the signature **`valueOf(String)`**

`valueOf` returns the constant that corresponds to the argument string

The valueOf method

example using **Month**:

```
Month m = Month.valueOf("MAY");
System.out.println(m == Month.MAY);
```

prints:

```
true
```

Constant-specific methods

It's possible each constant in an enumeration to have its own version of a method

- declare an abstract method in the enum
- each constant then overrides the abstract method in a constant-specific class body

consider creating an enum that represents the different arithmetic operations of a calculator

see Effective Java 3rd Edition, Item 34

```
public enum Operation {
  PLUS {
    @Override
    public double apply(double x, double y) {  return x + y;  }
  },
  MINUS {
    @Override
    public double apply(double x, double y) {  return x - y;  }
  },
  TIMES {
    @Override
    public double apply(double x, double y) {  return x * y;  }
  },
  DIVIDE{
    @Override
    public double apply(double x, double y) {  return x / y;  }
  };

  public abstract double apply(double x, double y);
}
```

Prefer enums to Boolean parameters

booleans are often used as parameters when there is a choice between two different values

consider the following hypothetical constructors:

`BinaryDigit(boolean value)`

true for 1, false for 0

`Thermometer(boolean scale)`

true for degrees Celcius, false for degrees Fahrenheit

`Multimeter(boolean mode)`

true for voltmeter, false for ammeter

Prefer enums to Boolean parameters

using enums instead of boolean values makes your code much more readable

```
new BinaryDigit(BinaryValue.ONE) or
new BinaryDigit(BinaryValue.ZERO)

new Thermometer(TemperatureScale.CELCIUS) or
new Thermometer(TemperatureScale.FAHRENHEIT)

new Multimeter(Mode.VOLTS_DC) or
new Multimeter(Mode.VOLTS_AC) or
new Multimeter(Mode.AMPS) or
new Multimeter(Mode.OHMS) or ...
```

When to use an enum

use an enum when you need a set of constants whose values are known at compile time