# Object-Oriented Programming Fundamentals

## Lecture/Workshop (Week 12)

### An Introduction to Enumerated Types

*Note: After the workshop, you can make a directory in your latcs account for this worksheet and cd into it. All files in this worksheet can be copied from the library area into your current directory using:*

```
cp  /home/1st/csilib/cse1oof/ws11/*  .
```

---

Sometimes a piece of data can take on a limited range of predetermined values. For example, the days of the week are limited to Monday, Tuesday, Wednesday, Thursday, Friday, Saturday and Sunday. What type of variable or attribute is most appropriate for storing this type of data?

A character?

- 'M' – Monday
- 'T' – Tuesday
- 'W' – Wednesday
- 'H' – Thursday
- 'F' – Friday
- 'S' – Saturday
- 'U' – Sunday

But some week days start with the same letter, and therefore some have to be represented by a non-intuitive value.

A char variable or attribute could also be mistakenly set to an invalid value.

A String?

- "Monday"
- "Tuesday"
- "Wednesday"
- "Thursday"
- "Friday"
- "Saturday"
- "Sunday"

With strings each day has a unique representation but they use more space than necessary.

A String variable or attribute could still be mistakenly set to an invalid value.

An integer?

- 1 – Monday
- 2 – Tuesday
- 3 – Wednesday
- 4 – Thursday
- 5 – Friday
- 6 – Saturday
- 7 – Sunday

These values may not be intuitive.

An integer variable or attribute could still be mistakenly set to an invalid value.

Ideally we would like a type that can only take on one of the allowed values and is intuitive to use. An enumerated type is a data type that allows a variable of that type to take on only a fixed set of values.

The allowed values are specified when the type is defined:

```
enum WeekDay {MONDAY, TUESDAY, WEDNESDAY,
              THURSDAY, FRIDAY, SATURDAY, SUNDAY}
```

We can create a variable or attribute of an enumerated type:

```
WeekDay today;
```

A variable of this type can then only take on one of the values defined in the enum.

To specify a value we must use the name of the enum, the dot operator and the value:

```
today = WeekDay.WEDNESDAY;
```

We can compare enumerated type values using the .equals() method or the == operator:

```
if (today == WeekDay.FRIDAY)
```

We can display an enumerated type value:

```
System.out.println(today);
```

would output the text

```
WEDNESDAY
```

An enum can be declared where it is used or it can be placed in a separate file when it is to be used by a number of classes. When it is placed in a file on its own, the filename follows the java convention.

The **`WeekDay`** enum would be in a file called *WeekDay.java* and would just contain the following declaration:

```
public enum WeekDay
{
   MONDAY,
   TUESDAY,
   WEDNESDAY,
   THURSDAY,
   FRIDAY,
   SATURDAY,
   SUNDAY
}
```

## Task 1

The following example declares an **`enum`** where it is used.

```
public class Weather
{
   enum Temperature {HOT, WARM, MILD, COOL, COLD}

   public static void main(String[] args)
   {
      Temperature currentTemp = Temperature.HOT;

      if (currentTemp.equals(Temperature.HOT))
      {
         System.out.println("Phew! It's " + currentTemp +
                            " today");
      }
      else
      {
         System.out.println("It's not too hot today, it's "
                             + currentTemp);
      }
```

```
        currentTemp = Temperature.WARM;

        if (currentTemp == Temperature.HOT)
        {
           System.out.println("Phew! It's " + currentTemp +
                              " today");
        }
        else
        {
           System.out.println("It's not too hot today, it's "
                              + currentTemp);
        }
    }
}
```

What is the output from the program?

## Enumerated types and the switch statement

We can use an enumerated type as the controlling expression in a switch statement. The enumerated type's values are then used as case labels. When the enumerated type's values are used as case labels in a switch statement, the `enum` name does not need to be specified. It is already known from the type of the switch's condition variable.

```
enum Temperature {HOT, WARM, MILD, COOL, COLD}

...

Temperature todaysTemp = ...;
switch (todaysTemp)
{
   case HOT:
   case WARM:
      System.out.println("Go to the beach");
      break;
   case MILD:
   case COOL:
      System.out.println("Go for a walk");
      break;
   case COLD:
      System.out.println("Stay at home");
      break;
   default:
      System.out.print("Programming error - ");
      System.out.println("this message should not be seen");
      System.exit(1);
}
```

### The ordinal() method

Enumerated types are really a type of class and as such, have methods associated with them. Also, each possible value an enumerated type can take is an object of the `enum`.

The ordinal value of an enumerated type's value is its position in the enumerated type's list of values. In the **`Temperature`** example the ordinal values for the temperatures are:

```
HOT = 0
WARM = 1
MILD = 2
COOL = 3
COLD = 4
```

By default the possible values are numbered in the order that they are specified. The ordinal values start at 0.

### Task 2

What is output by the **`Weather2`** program?

```
public class Weather2
{
   enum Temperature {HOT, WARM, MILD, COOL, COLD}

   public static void main(String[] args)
   {
      Temperature temp = Temperature.HOT;
      System.out.println(temp.ordinal() + " " + temp);
      temp = Temperature.WARM;
      System.out.println(temp.ordinal() + " " + temp);
      temp = Temperature.MILD;
      System.out.println(temp.ordinal() + " " + temp);
      temp = Temperature.COOL;
      System.out.println(temp.ordinal() + " " + temp);
      temp = Temperature.COLD;
      System.out.println(temp.ordinal() + " " + temp);
   }
}
```

---

### The compareTo() method

The **`compareTo()`** method returns a value based on the order of the values in the enumerated type.

### Task 3

What is output by the **`Weather3`** program?

```
public class Weather3
{
   enum Temperature {HOT, WARM, MILD, COOL, COLD}

   public static void main(String[] args)
   {
```

```
        Temperature t1 = Temperature.HOT;
        Temperature t2 = Temperature.COOL;
        Temperature t3 = Temperature.WARM;
        Temperature t4 = Temperature.WARM;

        comparison(t1, t2);
        comparison(t2, t3);
        comparison(t3, t4);
        comparison(t4, t1);
        comparison(t4, Temperature.MILD);
    }

    public static void comparison(Temperature temp1, Temperature temp2)
    {
        if (temp1.compareTo(temp2) == 0)
        {
            System.out.println(
                "The temperatures are the same, they are both: " +
                temp1);
        }
        else if (temp1.compareTo(temp2) < 0)
        {
            System.out.println(temp1 + " is hotter than " + temp2);
        }
        else
        {
            System.out.println(temp1 + " is colder than " + temp2);
        }
    }
}
```

---

### The valueOf() method (a class method)

The **`valueOf()`** method takes a **`String`** value and returns the enumerated type value associated with that **`String`**.

The **`String`** value must match exactly the enumerated type value.

```
public class Weather4
{
    enum Temperature {HOT, WARM, MILD, COOL, COLD}

    public static void main(String[] args)
    {
        Temperature t1 = Temperature.valueOf("HOT");
        System.out.println(t1);

        Temperature t2 = Temperature.valueOf("Cold");
        System.out.println(t2);
    }
}
```

The output of the **`Weather4`** program is:

```
HOT
Exception in thread "main" java.lang.IllegalArgumentException: Cold
        at Weather4$Temperature.valueOf(Weather4.java:3)
        at Weather4.main(Weather4.java:9)
```

---

Remembering that an **enum** is really a special type of class that has a fixed number of instances – one for each of its possible values

### The values() method (a class method)

Each possible value of an enumerated type is an instance object of the **enum**

Each enumerated type has a **values()** method that returns an array of objects of that type – one for each possible value

Consider the enumerated type **Gender** to represent the different types of language gender in the German language

```
public enum Gender {MASCULINE, FEMININE, NEUTER}
```

Calling the values method on the `Gender` enum would return an array containing the three objects of the **Gender** enum

```
public static void main(String[] args)
{
   Gender[] possibleGenders = Gender.values();

   System.out.println("The possible Genders are:");
   for (int i = 0; i < possibleGenders.length; ++i)
   {
      System.out.println(possibleGenders[i]);
   }
}
```

```
The possible Genders are:
MASCULINE
FEMININE
NEUTER
```

Remember the **ordinal()** method of enumerated types discussed earlier. The ordinal value of an enumerated type's value is its position in the enumerated type's list of values.

In the **Gender** example the ordinal values for the genders are:

```
MASCULINE = 0
FEMININE = 1
NEUTER = 2
```

We can now see a relationship between the ordinal value and the index in the **values** array for each possible value of the enumerated type.

## Adding Information to enums

Sometimes we may want to store more information within an **enum** than just the type's names.

As an **enum** is really a special type of class that has a fixed number of instances, it can be given additional attributes and methods.

Note that the constructor in an **enum** is not given an access modifier - it cannot be made public as code outside the **enum** cannot create new instances of it.

## Task 4

```
public enum MultiBirth
{
   TWINS (2),
   TRIPLETS (3),
   QUADRUPLETS (4),
   QUINTUPLETS (5),
   SEXTUPLETS (6);

   private int numberOfBabies;

   MultiBirth(int number)
   {
      numberOfBabies = number;
   }
   public int getNumber()
   {
      return numberOfBabies;
   }
}
--------------------------------
public class Nursery
{
   public static void main(String[] args)
   {
      MultiBirth[] mbValues = MultiBirth.values();

      for (int i = 0; i < mbValues.length; ++i)
      {
         System.out.println(mbValues[i].ordinal() + " " +
                            mbValues[i].toString() + " " +
                            mbValues[i].getNumber());
      }
   }
}
```

What is the output from this program?

---

*NOTE: You will NOT be asked to produce code using enums in any assessments in CSE1OOF or CSE4OOF. You may be expected to be able to understand simple code given to you and work out what that code does, as in the previous tasks.*

**Summary of methods discussed**

*Instance methods*

The toString() method
**`public String toString()`**
Returns the value as a String

The equals() method
**`public boolean equals(enum_type value)`**
(We can also use == with enumerated type values)

The ordinal() method
**`public int ordinal()`**
Returns the position of the value in the list of values (starting at 0)

The compareTo() method
**`public int compareTo(enum_type value)`**
Returns a positive, zero or negative number by comparing using the position of the value in the list of values

*Class methods*

The valueOf() method
**`public static enum_type valueOf(String name)`**
Returns the enumerated type value corresponding to name (exact match needed)

The values() method
**`public static enum_type[] values()`**
Returns an array containing the values of the enumerated type (in the order they are listed in the definition of the enumerated type)