8. Using Enums

Exam Objectives

1. Create and use enumerations with fields, methods and constructors

8.1 Purpose of enums

Enum is short for enumeration. An enumeration is a just way to mention a fixed number of things one by one. For example, if you are trying to express the state of a light switch, there are only two options - on and off. Similarly, if you are trying to work with the names of the days of a week, then there are only seven names - Monday to Sunday. This information is already available at the time of writing the code. Compare this with, for example, an Account entity. You don't know how many accounts can be there or what are their account numbers going to be at the time of writing the application. This information is available only at the run time. Thus, you will create Account instances as and when required at run time. This is what regular Java classes are for and you will model Account entity as a Java class.

An enum, on the other hand, is a special way of defining a class, which can be used when you already know the exact number of instances of that class at compile time and when you want to refer to those instances by predefined names at run time. Among other things, which I will show shortly, it lets you get away without creating and managing instances of that class explicitly.

8.2 Creating and using an enum

8.2.1 Enum syntax

In its most simple form, an enum can be defined like this:

```
enum AccountType{
  CHECKING, SAVINGS, FIXEDDEPOSIT;
}
```

You have the `enum` keyword, the enum name, and an enumeration of the instances of the enum that you want. The above enum definition defines an `AccountType` enum with three named instances - `CHECKING`, `SAVINGS`, `FIXEDDEPOSIT`. I have used uppercase for these names because, conventionally, names of the things that are constant are typically in upper case and these names are indeed constants. These names are like three final static variables of type `AccountType` that point to three unique `AccountType` instances.

The above enum can be used just like any other class. Here is a complete executable example:

```
enum AccountType{ CHECKING, SAVINGS, FIXEDDEPOSIT; }

class Account{
  String acctId;
  AccountType acctType;
  public Account(String acctId, AccountType acctType){
    this.acctId = acctId; this.acctType = acctType;
  }
}
```

```
void main(){
  Account a = new Account("1234", AccountType.CHECKING);
  System.out.println(a.acctType);
}
```

The above code prints `CHECKING`.

8.2.2 Features of an enum

Although the previous example of the `AccountType` enum is very minimal, an enum can be as feature rich as a regular class. It can have fields, constructors, and methods. Let me go over these details now.

Enum class

An enum **implicitly extends** `java.lang.Enum` class. This implies that you cannot make your enum extend from another enum or class. However, an enum can **implement interfaces**.

Although not important for the exam, an enum is either implicitly **final** or implicitly **sealed**, which basically means you cannot extend an enum using the extends clause but you may create anonymous subclasses for specific enum constants. Confusing? I know. Don't worry about it.

Enum fields

The first line inside an enum must be a list of names by which you want to refer to your enum instances. These will be the only instances of this enum that will ever be created. In the `AccountType` enum, I have specified three names. These three names are really three public, static, and final fields of type `AccountType` in the `AccountType` enum.

Besides these implicit fields, you may define any number of and any type of fields in an enum. For example:

```
enum AccountType{
  CHECKING, SAVINGS, FIXEDDEPOSIT; //these are implicitly public, static, and final
  private String description = "some description";
  private static final long serialVersionUID = 1L; //whatever
}
```

A natural question that arises now is how would these extra fields be initialized and accessed? Using constructors and methods, of course.

Enum constructors and methods

Just like a regular class, an enum can have one or more constructors. But enum **constructors** are **always private**. You cannot make them public or protected. If an enum has no constructor declaration, then a private constructor that takes no parameters is automatically provided by the

compiler. Now, what is the point of having a private constructor in an enum? Well, remember that you cannot create instances of enums willy-nilly. All instances of an enum are created automatically by the JVM. But the JVM uses the same constructors that you define in the enum while creating those instances. Here is an example:

```
enum AccountType{
   CHECKING("Checking account"), //using the String constructor
   SAVINGS("Savings account"), //using the String constructor
   FIXEDDEPOSIT; //using the no-args constructor

   private String description = "some desc";

   AccountType(){ } //implicitly private

   AccountType(String desc){ //implicitly private
      this.description =  desc;
   }
   public void setDescription(String s){
      this.description = s;
   }
   public String getDescription(){
       return this.description;
   }
}
```

In the above code, I have provided two constructors. One that doesn't take any argument and one that takes one String argument. Observe how these constructors are used while specifying enum constants.

I have also created public accessor methods for the `description` field. These methods can be invoked just like you invoke methods on any class instance. For example:

```
void main(){
  AccountType.FIXEDDEPOSIT.setDescription("Fixed deposit account");
  System.out.println(AccountType.FIXEDDEPOSIT.getDescription());
}
```

Observe that the `setDescription` method updates the description field of the enum. This shows that enums may not necessarily be immutable. They may have a mutable state.

Utility methods of an enum

An enum gets a few static and instance methods which come in handy while working with enums. Here is a list of these methods:

Static methods of an enum

The compiler provides an enum with two **public static** methods automatically:

1. A `values()` method that returns an array of its constants. For example: `AccountType[] atypes = AccountType.values();`
2. A `valueOf(String )` method tries to match the String argument exactly (i.e. in case-sensitive fashion) with an enum constant and returns that constant if successful, otherwise it throws `java.lang.IllegalArgumentException`. For example: `AccountType checkingAcctType = AccountType.valueOf("CHECKING");`

Instance methods of an enum

1. A public `toString()` method that returns the enum name as a String but you can override it to return anything you want.
2. A public `ordinal()` method, which returns the index (starting with 0) of that enum constant, i.e., the position of that constant in its enum declaration.
3. A `name()` method, which returns the name of this enum constant, exactly as declared in its enum declaration.
4. An `equals(Object )` method but this methods compares two enums using the `==` operator. If you want to customize the comparison, you may override it.
5. A `compareTo(E )` method that compares an enum constant with the specified object as per their order in declaration. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Enum constants are only comparable to other enum constants of the same enum type. The natural order implemented by this method is the order in which the constants are declared.

The following are a few more important facts about `java.lang.Enum` which you should know:

1. All enums implement `java.lang.Comparable` (thus, an enum can be added to sorted collections such as `SortedSet, TreeSet`, and `TreeMap`). The natural order of the enum values is the order in which they are defined, i.e., in the order of their ordinal value.
2. Since an enum maintains exactly one instance of its constants, you cannot clone it. You cannot even override the clone method in an enum because `java.lang.Enum` makes it final.

8.2.3 Benefits of enums

1. The most important benefit of enums is that they promote type-safety in code. Without enums, you would be forced to use some other data type such as `int` or `String` to represent your fixed set of values. However, there is no easy way to restrict the values these data types can store. For example, if you choose an `int` to represent `AccountType`, there is no way to restrict anyone from using values other than 0, 1, and 2 for account type. You will have to write quite a lot of code to provide this protection. In fact, there is a name for the kind of code that you would have to write for this purpose. It is called **"typesafe enum pattern"** and it is after the popularity of this coding pattern that enums were introduced in Java 1.5. In other words, enums are a readymade standard solution for the typesafe enum pattern in the Java world. It is not important to go over this pattern here but you should read about it if you have time.

2. Since enums are as feature rich as regular classes, they allow you to have attributes and methods along with static values and this makes the code closer to OOP.

3. As you will see in the Using Decision Constructs chapter, it is easier to write exhaustive switch statements and expression with enums because they have a fixed and known number of values.

4. Since the `java.lang.Enum` class implements `java.io.Serializable`, you get efficient serialization automatically with enums.

8.2.4 Quiz

Consider the following enum definition.

```
enum AccountType{
   CHECKING(0), SAVINGS(1);
   private int acctTypeId;
   AccountType(int id){ this.acctTypeId = id; }

   //INSERT CODE HERE
}
```

Which of the following pieces of code can be inserted individually in the above enum definition?
Select 2 correct options.

A.

```
AccountType getAccountType(int id){
    return values()[id];
}
```

B.

```
void newAccountType(String name, int id){
    if(id>1) values().add( new AccountType(name, id) );
}
```

C.

```
void newAccountType(int id){
    if(id>1) values().add( new AccountType(id) );
}
```

D.

```
void newAccountType(String name, int id){
    if(id>1) values.put(name, new AccountType(id) );
}
```

E.

```
AccountType getAccountType(int id){
    return new AccountType(id);
}
```

F.

```
FIXEDDEPOSIT(2);
```

G.

```
@Override
public String toString(){ return "AccountType enum"; }
```

H.

```
public static AccountType[] values(){ return super.values(); }
```

Correct answer is A and G.

Remember that instantiating an enum explicitly in anyway is not allowed. Therefore, any code that does `new` on `AccountType` is wrong straightaway.

Every enum gets a `values()`method. It returns an array of enum instances defined in that enum. Since it returns an array, it is possible to apply an index on it to access a specific element of that array as done in **option A**. It is, therefore, valid. Of course, it may throw an `ArrayIndexOutOfBoundsException` at run time if an invalid index is applied.

Option F is invalid because it tries to define an enum constant on a separate line than the first one. All enum constants must be defined on the first line inside the enum definition separated by commas.

Every enum inherits a `toString()` method. You are allowed to override it. Therefore, **Option G** is valid. Not relevant here, but the same goes for `equals(Object )` and `hashCode()` methods as well. On the other hand, enums also inherit an instance method named `ordinal()` from its super class `java.lang.Enum` but this method is final and so, you cannot override it.

Every enum is provided with `values()` and `valueOf(String )` static methods by the compiler implicitly. You are not allowed to define them explicitly in the enum. Therefore, **option H** is invalid.

8.3 Exercise

1. Create an enum named `Coin` having five constants - PENNY, NICKEL, DIME, QUARTER, and DOLLAR.
2. Separately, write a loop that iterates over `Coin` and prints the name and index of each element in `Coin`.
3. Create an `int` field named `value` in `Coin`. Initialize it using an `int` value supplied via `Coin`'s constructor. Enhance the loop to print this value as well.
4. Create a static method named `getCoin(int value)` that returns a `Coin` for a given value. Return `null` if there is no coin for the given value.

5. Create an instance method named `add(Coin )` that takes a `Coin` as an argument and returns the sum of their values.

6. Change the add method to return a `Coin` if the sum matches the value of any other `Coin` and `null` otherwise.