OBJECTIVES

After completing “Enumerated Types,” you will be able to:

• Use native enumerated types, or enums, to identify sets of related constant values.
• Convert enums to and from string and ordinal representations.
• Encapsulate state and behavior in enumerated value objects.
Static Finals as Enumerated Types

- Before Java 5, the best way to capture sets of related constants was to use `static final` fields:

```java
public static final int CMYKCOLOR_CYAN = 0;
pUBLIC static final int CMYKCOLOR_MAGENTA = 1;
pUBLIC static final int CMYKCOLOR_YELLOW = 2;
pUBLIC static final int CMYKCOLOR_BLACK = 3;
```

- This was considered good practice because it defined each value as a unique identifier.
  - Compared to just using the numbers 0, 1, 2, and 3, use of the above field names is descriptive.
  - It also keeps mistypes and misuses to a minimum, as the compiler can check for correctness.

- But this approach falls short of defining a distinct type in the type system for the enumeration.
  - These are ultimately just compiler-checkable names for integers, or strings, or some other general-purpose type.

```java
void processLayer (int color, int[][] layer) {
    if (color != CMYKCOLOR_BLACK)
        applyColorAdjustment (layer);
    ...
}
```

  - What if someone, for some reason, tried this ...

```java
if (color != 4)
    applyColorAdjustment (layer);
```

  - ... or passed 4 as an argument to the method?
Enumerated Types

- Java 5 introduces a native **enumerated type**.
- This allows an application to define a finite set of values that are useful in its work.
  - The enumeration itself, and then each value, can be given a unique name, rather than an arbitrary integer or string value.

```java
enum CMYKColor { CYAN, MAGENTA, YELLOW, BLACK }
```
  - These names and the type itself can be checked by the compiler, where an integer or string could not.
  - So code that mistypes or misuses a value will be flagged with a compile-time error, rather than passing the compiler and then failing at runtime.

```java
void processLayer (CMYKColor color, int[][] layer) {
    if (color != CMYKColor.BLACK)
        applyColorAdjustment (layer);
    ...
}
```
- This is known as **type-safe enumeration**.
  - The enum is a distinct type, and there is a fixed set of instances of that type representing all of the values.
  - There is no way to create other instances.
- An enum can be defined at the “top level” – that is, in its own source file as peer of public classes.
  - It is also commonly defined as a member of a class that uses it, in which case it is automatically a **static inner class**.
Advantages of Enumerated Types

- Enumerated types improve on this in a few ways.

```java
enum CMYKColor { CYAN, MAGENTA, YELLOW, BLACK);
```

- They facilitate natural **scoping** of identifiers: we don’t have to back qualifying prefixes into our own identifiers in order to have descriptive symbols in our code.

```
CMYKColor.BLACK vs. CMYKCOLOR_BLACK
```

- We can **iterate** over an enumerated type to work on all values.
- Each value has a natural and unique **string representation**.
- There is exactly one instance of each value, so we can compare using the `==` operator, rather than the `equals` method.
- Each value is actually an object, and the enumeration is actually the class of that object. Thus, enumerated types can be **stateful** and even **behavioral**, much like ordinary Java classes.
**String Representations**

- Every value of an enumeration has a string representation, which is just the Java identifier mapped character-for-character into a `String` instance.
- We can get the string representation either with the `name` method (unique to `enums`) or the `toString` method (defined on `Object`, implemented for `enums`).

```java
System.out.println (thisValue.name ());
System.out.println (thatValue);
```

- We can go the other way, too, using the static method `valueOf`:
  - This method throws the `IllegalArgumentException` if the given value is not found in the enumeration.
  - So it’s often couched in exception handling:

```java
Color color = null;
try {
    color = Color.valueOf ("CYAN");
} catch (IllegalArgumentException ex) {
    System.out.println ("That's not a legal value!");
}
```
Looping and Enumerated Types

- The `enum` type defines a finite set of named values.
- What if we wanted to loop over all those values?
- Any enumerated type provides a static method, `values`, which returns an array containing all its possible values, in the order of their declaration.
- This then is another good application for the simplified `for` loop:

```java
System.out.print("Colors are ");
for (Color color : Color.values())
    System.out.print(color.name() + " ");
```
Color Test

• In Examples/Color/Step1, we have exactly the CMYKColor enumeration that we’ve been considering so far, defined as a public, top-level type.
  – See src/cc/color/CMYKColor.java:

```java
public enum CMYKColor {
    CYAN, MAGENTA, YELLOW, BLACK;
}
```

• A simple test application exercises this type.
  – See src/cc/color/ColorTest.java:

```java
public static void main (String[] args) {
    CMYKColor color = CMYKColor.MAGENTA;
    int ordinal = 1;
    // Wouldn't compile -- incompatible types:
    // ordinal = (int) color;
    // color = (CMYKColor) ordinal;
    ordinal = color.ordinal ();
    String colorName = color.name ();
    System.out.println("Ordinal position of " + colorName + " is " + ordinal);
}
```
Color Test

We convert a user-provided string to the corresponding value object – or advise the user that we can’t:

```java
if (args.length != 0)
    try {
        CMYKColor yours = CMYKColor.valueOf (args[0]);
        System.out.println ("Ordinal position of " + 
                            args[0] + " is " + yours.ordinal ());
    }
    catch (IllegalArgumentException ex) {
        System.out.println ("  (CMYK colors are " + 
                             "CYAN, MAGENTA, YELLOW, and BLACK."");
    }
```

Finally, we write the names of all the colors to the console:

```java
System.out.println ();
System.out.println ("All colors: ");
for (CMYKColor each : CMYKColor.values ())
    System.out.format ("  %-7s\n", each);
```
**Color Test**

- Build and test the application as follows:

  build  
  run ColorTest  
  Ordinal position of MAGENTA is 1

  All colors:  
  CYAN  
  MAGENTA  
  YELLOW  
  BLACK

  run ColorTest YELLOW  
  Ordinal position of MAGENTA is 1  
  Ordinal position of YELLOW is 2

  All colors:  
  CYAN  
  MAGENTA  
  YELLOW  
  BLACK

  run ColorTest YELL  
  Ordinal position of MAGENTA is 1  
  CMYK colors are CYAN, MAGENTA, YELLOW, and BLACK.

  All colors:  
  CYAN  
  MAGENTA  
  YELLOW  
  BLACK
**Transaction Types**

**Suggested time: 30 minutes**

In this lab you will refactor a small application that processes transactions on bank accounts. Transactions are already encapsulated as objects, with each transaction having a type and an amount. But the transaction type is just a string, which is not type-safe and can’t be processed in any special way. You’ll refactor the application to use a new `enum`.

Along the way you’ll also convert to using formatting for the application’s console output.

Detailed instructions are found at the end of the chapter.
A Bank of Accounts

Suggested time: 30 minutes

In this lab you will complete an enhancement to the Bank test application by which it can hold a list of accounts, each with a unique ID. It can then test them selectively by ID and can show the final state of the whole system when done.

You’ll review this code, and then sweeten up the interface to one of the helper methods by supporting a variable parameter list, and changing the calling code to take advantage of that change.

This lab is marked optional because it is really review in using varargs, and not so much about enumerated types.

Detailed instructions are found at the end of the chapter.
Understanding Enumerated Types

• Thus far we’ve used enumerated types mostly as simple sets of unique names.
• In fact, the members of these sets are not simple values; they are objects.
• An enum is actually a specialized sort of class – and an interesting exercise in controlling object creation.
• The compiler translates an enum definition into a class of the given name that defines a set of static final members, whose names are those given in the enum definition.
  – This class closely follows a design pattern known as the flyweight, in which objects are desired but there can only be a finite set of possible object states.
  – Thus it is worthwhile to limit the number of possible instances to one for each possible value.
  – A flyweight hides its constructor, and rather than provide a factory method it simply offers a full set of public, pre-constructed instances, ready to use.
  – All callers share these instances – there is only one CMYKColor.CYAN object, which is fine because it’s immutable anyway.
Disassembling CMYKColor

- It may be illuminating to see a disassembly of an enum.
- Let’s investigate the CMYKColor enum defined in Examples/Color/Step1: build the project, and then set the class path to the build directory:

```java
build
set CLASSPATH=build
```

- Run the `javap` tool – this is a crude disassembler:

```
javap cc.color.CMYKColor
Compiled from "CMYKColor.java"
public final class cc.color.CMYKColor
    extends java.lang.Enum{
    public static final cc.color.CMYKColor CYAN;
    public static final cc.color.CMYKColor MAGENTA;
    public static final cc.color.CMYKColor YELLOW;
    public static final cc.color.CMYKColor BLACK;
    public static cc.color.CMYKColor[] values();
    public static cc.color.CMYKColor
        valueOf(java.lang.String
            static {};
    }
```

- Aha! There are many of the symbols we’ve been using as if they were native language features:
  - **CMYKColor.CYAN, CMYKColor.MAGENTA**, etc. – notice that these are instances of the **CMYKColor** class itself
  - The `values` array
  - The `valueOf` method that converts a string to an instance
Disassembling CMYKColor

- But, we’ve seen a few other methods – name, ordinal, etc.
- Where are they?
- Notice that CMYKColor is found to extend the class Enum.
- Okay ... let’s disassemble that!

javap java.lang.Enum
Compiled from "Enum.java"
public abstract class java.lang.Enum
   extends java.lang.Object
   implements java.lang.Comparable,
            java.io.Serializable{
      public final java.lang.String name();
      public final int ordinal();
      public java.lang.String toString();
      public final boolean equals(java.lang.Object);
      public final int hashCode();
      public final java.lang.Class
         getDeclaringClass();
      ...
}

- In fact the Enum class is a generic type – and a strange one even among generics, as its type parameter is the subclass that extends it:

public class Enum<? extends Enum>

   - We’ve not studied generics yet, so it’s hard to say more.
   - But the fact that Enum is generic offers some insight into how methods like name and ordinal can be defined on the base class, and even be final, while values are defined by the derived class.
Stateful and Behavioral Enumerations

- Seeing that an enum is actually a specialized class might get your creative juices flowing …
- Could an enumerated type define more than just an ordered list of values?
- Yes, and in this way Java enumerations are surprisingly powerful.
  - Enums can have state, beyond a single ordinal value.
  - They can define behavior – methods, just like any class.
- We now consider the full grammar of enum definitions, of which our uses so far have just been the simplest case:

```java
class Name {
  value1{(initializers)},
  value2{(initializers)}, ...
  valueN{(initializers)} ;;

  constructor! fields! methods! }
```

- Only the enum declaration itself, and the value list at the beginning, are treated differently by the compiler.
- From there out, ordinary member definitions will be compiled as usual.
- Let’s look at some of the possibilities this opens up.
Car Handling

In Examples/Cars/Step1, classes Car and UsedCar encapsulate cars for sale by a dealer:

Each class defines an enumeration to capture some factor relevant to its perceived value.
• **Car.Handling** defines handling rating and also associates a multiplier that affects the perceived value of the car during a test drive – see `src/cc/cars/Car.java`:

```java
public enum Handling {
    excellent (.2), good (.1), fair (0), poor (-.1);

    private Handling (double adjustment)
    {
        this.adjustment = adjustment;
    }

    public double getAdjustment ()
    {
        return adjustment;
    }

    private double adjustment;
};
```

• The compiler extrapolates this to a full-fledged flyweight class.
Car Handling

• Without the use of enumerated types, the testDrive method had to switch/case over possible handling values to derive the multiplier itself:

```
feedback += "Handling is " + handling + ";";
if (handling.equals (Handling.excellent))
    factor += .2;
else if (handling.equals (Handling.good))
    factor += .1;
else if (handling.equals (Handling.poor))
    factor -= .1;
```

• Now it can rely on getAdjustment instead of mapping enum member to value factor.

```
feedback += "Handling is " + handling + ";";
factor += handling.getAdjustment ();
```

  – And so can anyone else, which is the point of encapsulation: we get reuse of a mapping that’s inherent to the Handling type in the first place.
• In the same example, the UsedCar class takes a similar approach to the Condition enum.

• Here, though, we need state and behavior, because the impact on perceived value can’t be derived by simple arithmetic – see src/cc/cars/UsedCar.java:

```java
public enum Condition {
    excellent (1.25, "Seems to be in great shape."),
    good (1.0, "It's been well kept.")
    fair (.75, "Shows its age."),
    poor (.5, "What a mess!")
    wreck (0.0, "It's a wreck! ...");

    Condition (double multiplier, String feedback) {
        this.multiplier = multiplier;
        this.feedback = feedback;
    }

    public void adjustFeedback (TestDriveResults results) {
        results.setPerceivedValue (results.getPerceivedValue () * multiplier);

        if (this == wreck)
            results.setFeedback (feedback);
        else
            results.addFeedback (feedback);
    }

    private double multiplier;
    private String feedback;
};
```
Used-Car Condition

- Its override of `testDrive` simply invokes this encapsulated behavior:

```java
public TestDriveResults testDrive ()
{
    TestDriveResults results = super.testDrive ();
    condition.adjustFeedback (results);
    return results;
}
```

- Build and test the application as follows, to see the impact of handling and condition ratings:

```java
build
run list
ED9876: 2004 Toyota Prius (Black)
PV9228: 2004 Subaru Outback (Green)
BA0091: 2003 Ford Taurus (Gold)
HJ5599: 2004 Saab 9000 (Silver)
ME3278: 2003 Saturn Ion (Plum)
CZ7821: 1977 AMC Pacer (Blue) -- USED
AR7993: 1974 Ford Pinto (Dust) -- USED
RP5191: 1978 Renault Le Car (Yellow) -- USED
RG0504: 1991 Geo Metro (Midnight) -- USED
WQ0227: 1972 Ford El Camino (Blue and tan) -- USED

run drive ED9876
Feels a bit weak. Handling is fair.
Seems like this car is worth $21,599.09.

run drive AR7993
It's a wreck! Couldn't get it to move.
Seems like this car is worth $0.00.
Color Conversion

- In Demos/enum, we’ll use an enhanced CMYKColor enumeration that can help with conversion between CMYK and RGB color models.
  - The completed demo is found in Examples/Color/Step3.
- src/cc/color/CMYKColor.java has a lot more to it now:

```java
public enum CMYKColor {
    CYAN  (  0, 255, 255),
    MAGENTA (255,   0, 255),
    YELLOW  (255, 255,   0),
    BLACK   (  0,   0,   0);

    private int red;
    private int green;
    private int blue;

    private CMYKColor (int red, int green, int blue) {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public int getRed () {
        return red;
    }

    public Color convertToRGB () {
        return new Color (red, green, blue);
    }
};
```
Color Conversion

- But, so far, the application code doesn’t do much with the new state (red/green/blue values) or behavior (ability to pack up an AWT Color instance that represents the value object).

1. Open src/cc/color/ColorTest.java, and enhance the report on each color by showing the RGB values:

```java
for (CMYKColor each : CMYKColor.values())
    System.out.format("%-7s converted to RGB: (%3d, %3d, %3d)\n",
        each, each.getRed(), each.getGreen(), each.getBlue());
```

2. Build and test to see that you can pull individual property values out of the enumerated values:

```bash
build
demo run ColorTest
```

Ordinal position of MAGENTA is 1

All colors:
- CYAN converted to RGB: (0, 255, 255)
- MAGENTA converted to RGB: (255, 0, 255)
- YELLOW converted to RGB: (255, 255, 0)
- BLACK converted to RGB: (0, 0, 0)
Color Conversion

3. Try a second application that shows actual CMYK separations of color images:

run Viewer
Color Conversion

4. Click the different buttons and see the individual layers of the image:
Color Conversion

5. Open `src/cc/color/Viewer.java` to see how this works. Start in the constructor, where the GUI is assembled:

   ```java
   ButtonGroup group = new ButtonGroup () ;
   
   − For each color, we derive a user-friendly label by pushing all but the first letter of the string representation to lower case:
   ```java
   for (CMYKColor color : CMYKColor.values ())
   {
       String label = color.toString () .charAt (0) +
       color.toString () .toLowerCase () .substring (1);
   
   − Then we build a button with that label, and connect it to an event handler that will load a file whose name is partly based on the label as well:
   ```java
   JToggleButton button = new JToggleButton (label);
   button.addActionListener (new ChannelChanger
   (baseName + "_" + label + ".gif" ));
   group.add (button);
   pnTop.add (button);
   }
   ```

6. Add code to set the background color of each button to the color it represents:

   ```java
   JToggleButton button = new JToggleButton (label);
   button.setBackground (color.convertToRGB ());
   ```

7. The black button will not look right unless we treat it a little differently, so set the foreground color to white for that one:

   ```java
   if (color == CMYKColor.BLACK)
       button.setForeground (Color.white);
   ```
8. Build and test again, and see that the buttons are a little splashier:

```java
build
run Viewer Curiosity
```

9. You can read each of three different image sets, by the way – “Leaves” is just set as the default:

```java
run Viewer Leaves
run Viewer Curiosity
run Viewer Race
```
Transaction Fees

Suggested time: 45 minutes

In this lab you will complete the Bank case study by imposing fees for certain types of transactions – on accounts in certain standings. First, you’ll introduce overdraft protection by allowing transactions to create a negative account balance; in the process you’ll define an enum `Account.Standing` for no-fee (lots of money), good, overdrawn, and frozen (beyond overdrawn) status values. You’ll refuse all non-deposit transactions on frozen accounts.

Then you’ll impose transaction fees based on transaction type and account standing. This will involve adding state to the `Transaction.Type` enum and behavior to your new `Account.Standing` enum.

Detailed instructions are found at the end of the chapter.
SUMMARY

- Enumerated types are a natural part of programming for many OO-trained developers, and in a way it’s been a strange omission from the Java language for years.

- Java’s vision of enums as more than just values in a set is powerful, too: full realization of the flyweight pattern opens up a number of interesting design options.
Transaction Types

In this lab you will refactor a small application that processes transactions on bank accounts. Transactions are already encapsulated as objects, with each transaction having a type and an amount. But the transaction type is just a string, which is not type-safe and can’t be processed in any special way. You’ll refactor the application to use a new enum. Along the way you’ll also convert to using formatting for the application’s console output.

Lab workspace: Labs/Lab2A
Backup of starter code: Examples/Bank/Step1
Answer folder(s): Examples/Bank/Step2 (intermediate)
Examples/Bank/Step3 (final)
Files: src/cc/bank/Transaction.java
src/cc/bank/Account.java
src/cc/bank/Test.java

Instructions:
1. Review the starter code: two related classes Account and Transaction and a Test application.

2. Build and run the starter application, using the prepared scripts:
   build
   run

Creating new account with balance of $1200.00
After DEPOSIT, balance is $1700.0
After CHECK, balance is $900.0
After WITHDRAW, balance is $400.0
After posting interest, balance is $401.0

Creating new account with balance of $500.00
After DEPOSIT, balance is $1000.0
After CHECK, balance is $200.0
REFUSED WITHDRAW; reason is Insufficient funds.
After posting interest, balance is $200.5
3. You see that the transaction types are captured as strings. There are constant values defined on the class, which minimizes the likelihood of coding errors; but still this system is not type-safe. First, get rid of the four `static final` strings on the `Transaction` class.

4. In their place, create an enumeration called `Type`, with those four values as the members of the enumeration.

5. Change the type of the `type` field from `String` to `Type`, and change the constructor and the `getType` method accordingly.

6. Change code that uses the transaction type in both `Account` and `Test` classes:
   - Where you find references of the form `Transaction.DEPOSIT`, change to `Transaction.Type.DEPOSIT`.
   - Where you see the use of the type `String`, change this to `Transaction.Type`.
   - You must compare strings using the `equals` method, but each `enum` member is a singleton, so you can safely compare using the `==` operator.

7. You should be able to build and test again, with identical results.

```
build
run
Creating new account with balance of $1200.00
After DEPOSIT, balance is $1700.0
After CHECK, balance is $900.0
After WITHDRAW, balance is $400.0
After posting interest, balance is $401.0

Creating new account with balance of $500.00
After DEPOSIT, balance is $1000.0
After CHECK, balance is $200.0
REFUSED WITHDRAW; reason is Insufficient funds.
After posting interest, balance is $200.5
```

... but the output is not very clean; let’s see about that.
8. Find the `System.out.println` calls in the Test methods `process` and `exercise`. Replace these with calls to `System.out.format`, and instead of concatenating strings to get your output, devise a format string and then supply the appropriate number of arguments to match your formatting fields. When you build and test, you want your output to look like this:

```
build
run

Creating new account with balance of $1200.00
After DEPOSIT, balance is $ 1,700.00
After CHECK, balance is $ 900.00
After WITHDRAW, balance is $ 400.00
After post interest, balance is $ 401.00

Creating new account with balance of $500.00
After DEPOSIT, balance is $ 1,000.00
After CHECK, balance is $ 200.00
REFUSED WITHDRAW; reason is Insufficient funds.
After post interest, balance is $ 200.50
```

This is the intermediate answer in Step2.

**Optional Steps**

9. You can simplify the code a bit by using a static import: in each of the classes that uses `Transaction`, add this statement:

```
import cc.bank.Transaction.Type;
```

10. You can simplify the code a bit by importing the `Type` enumeration: in each of the classes that uses `Transaction`, add this statement:

```
import cc.bank.Transaction.Type;
```

11. Now, though you must still refer to the deposit type as `Type.DEPOSIT`, you can at least leave off the `Transaction` qualifier. Make this change throughout both source files, build, and test once again: output should be the same as in your previous test.

   This is the final answer in Step3.
A Bank of Accounts

In this lab you will complete an enhancement to the Bank test application by which it can hold a list of accounts, each with a unique ID. It can then test them selectively by ID and can show the final state of the whole system when done.

You'll review this code, and then sweeten up the interface to one of the helper methods by supporting a variable parameter list, and changing the calling code to take advantage of that change.

Lab workspace: Labs/Lab2B
Backup of starter code: Examples/Bank/Step4
Answer folder(s): Examples/Bank/Step5
Files: src/cc/bank/Test.java

Instructions:

1. The starter code for this lab is similar to the answer set for the previous lab, but a few enhancements have already been made. The Test class is now meant to be instantiated and holds an array of account objects. The main method now exercises the first three accounts, getting references to each one by calling getAccounts on the Test instance and reading a specific item from the array.

2. Build and test the starter code, and you should see:

```java
build
run
Testing account 1 ...
After DEPOSIT, balance is $ 1,700.00
After CHECK, balance is $ 900.00
After WITHDRAW, balance is $ 400.00
After post interest, balance is $ 401.00
Testing account 2 ...
After DEPOSIT, balance is $ 1,000.00
After CHECK, balance is $ 200.00
REFUSED WITHDRAW; reason is Insufficient funds.
After post interest, balance is $ 200.50
Testing account 3 ...
After DEPOSIT, balance is $ 2,900.00
After CHECK, balance is $ 2,100.00
After WITHDRAW, balance is $ 1,600.00
After post interest, balance is $ 1,604.00
```
3. Make the `process` method a tad more useful by introducing a variable parameter list. Add the ... characters after the `Transaction` parameter type, and change the parameter name to `xaList`.

4. Now wrap the existing method implementation in a loop over `xaList`. Use a simplified `for` loop and name your loop variable `xa`; that way the existing code doesn’t need to be changed at all.

5. Take advantage of the new feature by calling `process` only once from `exercise`, passing the account and then all three transactions in a row.

6. Build and test and your output should be unchanged from the previous test.

7. At the bottom of `main`, write a loop to summarize the state of all accounts, again using formatted output: show account ID and balance.

8. Build and test. You should now see output as shown below:

   build
   run

   Testing account 1 ...
   After DEPOSIT, balance is $ 1,700.00
   After CHECK, balance is $ 900.00
   After WITHDRAW, balance is $ 400.00
   After post interest, balance is $ 401.00

   Testing account 2 ...
   After DEPOSIT, balance is $ 1,000.00
   After CHECK, balance is $ 200.00
   REFUSED WITHDRAW; reason is Insufficient funds.
   After post interest, balance is $ 200.50

   Testing account 3 ...
   After DEPOSIT, balance is $ 2,900.00
   After CHECK, balance is $ 2,100.00
   After WITHDRAW, balance is $ 1,600.00
   After post interest, balance is $ 1,604.00

   Account 1 balance is $ 401.00
   Account 2 balance is $ 200.50
   Account 3 balance is $ 1,604.00
   Account 4 balance is $ 100.00
Transaction Fees

In this lab you will complete the Bank case study by imposing fees for certain types of transactions – on accounts in certain standings. First, you’ll introduce overdraft protection by allowing transactions to create a negative account balance; in the process you’ll define an enum `Account.Standing` for no-fee (lots of money), good, overdrawn, and frozen (beyond overdrawn) status values. You’ll refuse all non-deposit transactions on frozen accounts.

Then you’ll impose transaction fees based on transaction type and account standing. This will involve adding state to the `Transaction.Type` enum and behavior to your new `Account.Standing` enum.

**Lab workspace:** Labs/Lab2C

**Backup of starter code:** Examples/Bank/Step5

**Answer folder(s):** Examples/Bank/Step6 (intermediate)  
Examples/Bank/Step7 (final)

**Files:**  
src/cc/bank/Transaction.java  
src/cc/bank/Account.java  
src/cc/bank/Test.java

**Instructions:**

1. Open `Account.java` and add a new enumeration `Standing`, with values `NO_FEES`, `GOOD`, `OVERDRAWN`, and `FROZEN`.

2. Give this enumeration two private static final double fields: set `NO_FEE_THRESHOLD` to 1000 and `OVERDRAW_LIMIT` to -100.

3. Give the enumeration a public static method called `determineStanding`, which takes an `Account` and returns an instance of `Standing`. Implement it to return the appropriate value based on the account balance: if it’s over the `THRESHOLD` return `NO_FEES`; otherwise any balance over zero is in `GOOD` standing; otherwise a value above `OVERDRAW_LIMIT` is just `OVERDRAWN`, but under that limit it’s `FROZEN`.

4. Add a field `standing` to the `Account` itself, of type `Standing`, and add an accessor method `getStanding`.

5. In the `Account` constructor, call `Standing.determineStanding`, passing `this`, and use that to initialize `standing`.

6. In `executeTransaction`, change the rules about what transactions we refuse. First, remove the code that throws a `TransactionRefused` for insufficient funds.
7. At the top of the method, once the `xaType` has been derived, check to see if the account is **FROZEN** and `xaType` is anything other than **DEPOSIT**. If so, throw a `TransactionRefused`.

8. At the end of `executeTransaction`, update the account standing with code similar to what you put in the constructor.

9. In `postInterest`, assure that interest will only be credited to accounts in either **GOOD** or **NO_FEES** standing.

10. Build and test. Your results should be different now than in the previous lab:

    ```
    build
    run
    Testing account 1 ...
    After DEPOSIT, balance is $1,700.00
    After CHECK, balance is $900.00
    After WITHDRAW, balance is $397.50
    After post interest, balance is $398.49
    Testing account 2 ...
    After DEPOSIT, balance is $998.50
    After CHECK, balance is $197.75
    After WITHDRAW, balance is $-304.75
    After post interest, balance is $-304.75
    Testing account 3 ...
    After DEPOSIT, balance is $2,900.00
    After CHECK, balance is $2,100.00
    After WITHDRAW, balance is $1,600.00
    After post interest, balance is $1,604.00
    Account 1 balance is $398.49; standing is GOOD
    Account 2 balance is $-304.75; standing is FROZEN
    Account 3 balance is $1,604.00; standing is NO_FEES
    Account 4 balance is $100.00; standing is GOOD
    This is the intermediate answer in Step6.
    ```
Transaction Fees

Optional Steps

11. Now let’s implement transaction fees. In `Transaction.Type`, define a private field `fee` of type `double`, and define an accessor `getFee`.

12. Define a constructor that takes a `double` and uses that to initialize `fee`.

13. Give each member of the enumeration a fee value: 2.5 for withdrawals, 1.5 for deposits, .75 for checks, and zero for the other two types.

Now, the logic to calculate a transaction fee involves both the transaction type (which encapsulates a base fee) and the account standing (which can either prohibit the transaction or modify the fee). Where should we encapsulate this logic?

It could go in `Transaction.Type`, which would give us a dependency on `Account.Standing`; in `Account.Standing`, which would give us a dependency on `Transaction.Type`; or in some external piece of code that would depend on both enumerated types.

There’s a case for any of the above options, but in order to explore the capabilities of enumerated types more fully, we’re going to go with the middle choice, and make `Account.Standing` able to report the fee for an account, given the transaction type.

14. In `Account.Standing`, add a method `getFee` that takes a transaction `Type` and returns a `Transaction` which will encapsulate the fee.

15. First, if this standing is `FROZEN` and the transaction type is other than `DEPOSIT`, throw an `IllegalStateException` – this is just defensive coding as this should never happen.

16. If the standing is `NO_FEES` or the transaction type is `ASSESS_FEE`, return `null`.

17. Get the base fee from the given transaction type.

18. If the standing is `OVERDRAWN` and the transaction isn’t a `DEPOSIT`, multiply the fee by 2.5.

19. Return a new `Transaction` of type `ASSESS_FEE`, with the fee amount.

20. Now, in `executeTransaction`, before executing the transaction itself, get the associated fee by calling `standing.getFee`. Store that in a local variable.

21. At the end of the method, just before refreshing the account standing, impose the fee: if the transaction you stored is not `null`, call `executeTransaction` and pass it as the argument. (There will be no infinite recursion since with a type of `ASSESS_FEE` this second transaction will not spawn any further calls.)
22. With this new system in place, the fourth of four accounts in the Test class will be more interesting; add code to main to test it along with the other three.

23. Also, update the code that reports the status of all accounts to add the account standing to each listing.

24. Build and test, and you should have output like this:

```java
build
run
Testing account 1 ...
After DEPOSIT, balance is $ 1,700.00
After CHECK, balance is $   900.00
After WITHDRAW, balance is $   397.50
After post interest, balance is $   398.49

Testing account 2 ...
After DEPOSIT, balance is $   998.50
After CHECK, balance is $   197.75
After WITHDRAW, balance is $  -304.75
After post interest, balance is $  -304.75

Testing account 3 ...
After DEPOSIT, balance is $ 2,900.00
After CHECK, balance is $ 2,100.00
After WITHDRAW, balance is $ 1,600.00
After post interest, balance is $ 1,604.00

Testing account 4 ...
After DEPOSIT, balance is $   598.50
After CHECK, balance is $  -202.25
REFUSED WITHDRAW; reason is Account frozen.
After post interest, balance is $  -202.25

Account 1 balance is $   398.49; standing is GOOD
Account 2 balance is $  -304.75; standing is FROZEN
Account 3 balance is $ 1,604.00; standing is NO FEES
Account 4 balance is $  -204.25; standing is FROZEN

This is the final answer in Step7.