Anatomy of a Class Notes

OrderLine.java

This is a simple class. Mostly, this class is a "data class". A data class is a class whose behavior is nothing more than to act as a container for a set of data elements.

Note that product is effectively a read-only attribute, it can't be changed once it is set by the constructor.

The getPrice method makes it look like this class has a read-only price attribute. However, in this case the attribute is computed from the product and quantity.

Order.java

This class has some more interesting features.

There is an initializer block. The initializer is found on lines 27 to 39. The code in this block is run prior to execution of the constructor.

The attribute nextOrderNumber is static. That means that there is exactly one copy of this attribute that is shared among all instances of the Order class.

There is a static initializer block. This can be found on lines 51 to 59. This code will be run when the class is used for the first time.

in addition to the nextOrderNumber attribute there is also a nextOrderNumber static method. (This shows that methods and attributes occupy distinct "name spaces".) Similar to a static attribute, a static method executes without reference to an object. As a matter of style it is considered appropriate to reference this method as Order.nextOrderNumber(), rather than obj.nextOrderNumber() where obj is an object of class Order.

There are three constructors for the Order class. This is an example of the first form of polymorphism: the compiler picks the right constructor based on the number and types of the parameters.

Note also that the second and third constructor use this(...) as the first statement in the constructor. In the case of the Order(Customer, boolean) this calls the Order() constructor and then finishes the initialization. In the case of the Order(Customer) constructor, it calls the Order(Customer, boolean) constructor to do all the work. The this(...) call must be the first statement in the constructor.

This class illustrates complex object constructor. Construction of a new Order object will proceed as follows:

1. All attributes are initialized to zero or null.
2. Attributes initializers and initializer blocks are executed in lexical order as they appear in the class definition:

a) The price attribute is initialized via the Currency.USD.zero() expression.

b) The initializer block is executed, initializing the customer, orderNumber, prepaid, and lineItems attributes.

3. The chosen constructor is executed. Depending on which constructor is invoked, this may invoke other constructors.

Product.java

This is an example of a simple interface. This is a promise that any class implementing this interface will have two methods with the names and signatures as shown.

Customer.java

This is another interface. This one contains three getters. Even though there is no accessibility specified for the getters, method declarations in interfaces are always public. You can specify public but it is redundant.

This interface also contains a field declaration for UNKNOWN_CUSTOMER. This field is always public static final. The static in the declaration is redundant.

InternalCustomer.java

This is a class for an internal customer—one that is inside the organization.

The class declaration includes the clause “implements Customer”. This says that this class will implement the Customer interface. It is a compiler error if the class does not provide implementations of the methods promised in the interface declaration.

Note that the methods implementing the interface are introduced with the @Override annotation. There methods are not actually overriding anything. But, they are implementing a method that has been promised by an interface.

ExternalCustomer.java

This class is for an external (real) customer.

This is an abstract class that “implements Customer”. Since this class is abstract, it does not have to implement all the methods in the Customer interface. For those it doesn’t implement, the promised is passed on to its sub-classes. In this case, the getCreditLimit method is not provided here. That means that it must be provided by any class that extends ExternalCustomer.

Note that the constructor for ExternalCustomer is declared protected. This says that this constructor can only be used by a sub-class. Since this is an abstract class, that’s not particularly different from public.

CorporateCustomer.java

This class “extends ExternalCustomer”. This class inherits all the interface and implementation from ExternalCustomer.

This class has attributes, creditLimit—declared here—and name and address—declared in ExternalCustomer. But, name and address are not directly accessible here since they were declared private in ExternalCustomer.

Note the constructor for CorporateCustomer. The call on the first line—super(name, address)--is a request to invoke the corresponding constructor on ExternalCustomer—the super-class. When constructing a class that extends another class, a constructor on the super-class must be invoked to initialize the super-class. That must be the first thing that happens in the constructor. The only way this can be deferred is by specifying a this(…) clause. The this clause passes the responsibility to the next constructor in line.

If the first line of the constructor is super()—no parameters—that can be omitted. The compiler will insert the clause as the first thing in the constructor. If it is omitted and there is not a no parameter constructor for the super-class, that is a compilation error.

In addition to the required getCreditLimit method, inherited from the Customer interface, there are two additional methods, billForMonth and remind.

Person.java

This is an interface for a Person.

PersonalCustomer.java

This is another sub-class of ExternalCustomer. Note the header. This class both extends ExternalCustomer and implements Person. A class is only allowed a single super-class. However, in addition to its single super-class, a class can implement as many interfaces as needed.

There is an interesting effect in this class. There is no implementation of the getPerson method implied by the Person interface. However, the getName() method inherited from ExternalCustomer satisfies that interface. The getName() required by Customer and the getName() required by Person will map into the same method. You could supply a new implementation in this class. That implementation would be the implmentation for both methods.

Money.java

This is an interface for Money. Money is a complicated concept once you have to support multiple currencies. Money represents an abstract amount of money that may be in multiple Currencies.

Note the method getAmount(Currency). That method is intended to return the amount of Money when converted to the given Currency.

Currency.java

This is an enumeration—enum—for all the currencies understood by the software, nine in this case.

An enumeration is a class with some special behavior. This class will have exactly nine instances. The list of symbols at the front are the names of the instances. These names are constants in the Currency class. So Currency.USD is the name of the US Dollars instance of Currency.

A stated, this class will have exactly nine instances. The instances are created when the class is initialized. No other instances can be created because the constructor is implicitly private. There is a compiler generated static initializer that creates all the instances.

Since each instance of this class is unique, you can compare enumerations with == and get the right result. The result will be the same as if the instances are compared with .equals(…).

Enums can have methods and attributes as shown here, but that is not required. Enums can also have constructors. In that case, the construction parameters are passed as part of the name of the enumeration, for example, USD(1, 2).

One other interesting feature of enums: you can use a switch statement on an enumeration type.

MoneyCurrency.java

This represents an amount of money in a single currency. Note that this class implements the Money interface. An instance of MoneyCurrency can act as a Money. Presumably, there would be other possible classes implementing Money, for example, one that implements a “bag” that contains differing amounts of different Currencies.

This is basically a data class holding an amount and a Currency except for the getAmount method which has to do currency conversion. This is actually a fairly complicated action. The implementation given here is a placeholder.