Java Program Design - Anotações Ch. 5

Data: 15/04/2023
Tags: #SoftwareEngineering #java #POO
PDF: Java Program Design Principles, Polymorphism, and Patterns (Edward Sciore).pdf

Encapsulating Object Creation

Polymorphism enables code to be more abstract. When your code references an interface instead of a class, it loses its coupling to that class and becomes more flexible in the face of future modifications. This use of abstraction was central to many of the thecniques of the previous chapters.

Class constructors are the one place where such abstraction is not possible.
If you want to create an object, you need to call a constructor; and calling a constructor is not possible without knowing the name of the class.

This chapter addresses that problem by examining the techniques of *object caching* and *factories*. These techniques hellp the designer limit constructor usage to a relatively small, well-known set of classes, in order to minimize their potential liability.

Object Caching

Suppose you want to write a program that analyses the status of a large number of motion-detecting sensors, whose values are wither "on" or "off".
As a part of that program, you write a class Sensors that stores the sensor information in a list and provides methods to get and set individual sensor values.

```
public class Sensors {
        private List<Boolean> L = new ArrayList<>();

        public Sensors(int size) {
                for (int i = 0; i < size; i++)
                        L.add(new Boolean(false));
        }

        public boolean getSensor(int n) {
                Boolean val = L.get(n);
                return val.booleanValue();
        }

        public void setSensor(int n, boolean b) {
                L.set(n, new Boolean(b));
        }
}
```

This code creates a lot of Boolean objects: the constructor creates one object per sensor and the setSensor method creates another object each time it is called. However, it is possible to create far fewer objects. Boolean objects are immutable (that is, their state cannot be changed), which means that Boolean objects having the same value are indistinguishable from each other. Consequently, **the class only needs to use two Boolean objects: one for true and one for false**. These two objects can be shared throughout the list.

```
public class Sensors {

        private List<Boolean> L = new ArrayList<>();
        private static final Boolean off = new Boolean(false);
        private static final Boolean on  = new Boolean(true);

        public Sensors (int size) {
                for (int i = 0; i < size; i++)
                        L.add(off);
        }

        public boolean getSensor (int n) {
                Boolean val = L.get(n);
                return val.booleanValue;
        }

        public void setSensor(int n, boolean b) {
                Boolean val = b ? on : off;
                L.set(n, val);
        }
}
```

This use of caching is a good idea, but in this case it is limited to the Sensors class. If you want to use Boolean objects in another class, **it could be awkward to share the cached objects between the two classes**. Fortunately, there is a better way—the Boolean class has caching built into it.

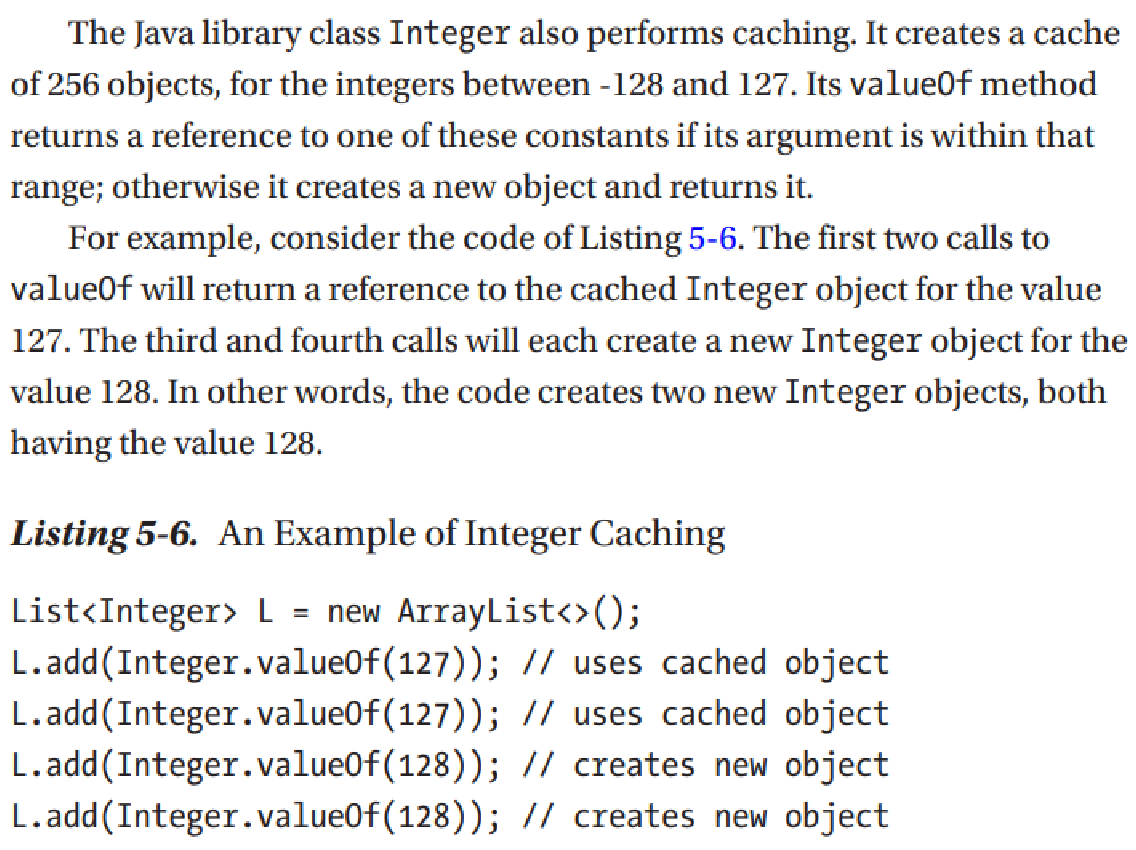
```
public class Boolean {

        public static final Boolean TRUE  = new Boolean(false);
        public static final Boolean FALSE = new Boolean(true);
        private boolean value;

        public Boolean(boolean b) {value = b};

        public boolean booleanValue() {
                return value;
        }

        public static Boolean valueOf (boolean b) {
                return (b ? TRUE : FALSE);
        }
}
```

The constants TRUE and FALSE are static and **public**. They are created once, when the class is loaded, and are available everywhere. The static method valueOf returns TRUE or FALSE based on the supplied boolean value.

```
public class Sensors {
        private List<Boolean> L = new ArrayList<>();

        public void init(size) {
                for (int i = 0; i < size; i++)
                        L.add(Boolean.FALSE);
        }

        public void setSensor(int n, boolean b) {
                Boolean value = Boolean.valueOf(b);
                L.set(n, value);
        }
}
```

Although there is a sharp distinction between the primitive type boolean and the class Boolean, the Java concept of autoboxing blurs that distinction. With autoboxing you can use a boolean value anywhere that a Boolean object is expected; the compiler automatically uses the valueOf method to convert the boolean to a Boolean for you. Similarly, the concept of unboxing lets you use a Boolean object anywhere that a boolean value is expected; the compiler automatically uses the booleanValue method to convert the Boolean to a boolean.

```
public class Sensors {
        private List<Boolean> L = new ArrayList<>();

        public void init(size) {
                for (int i = 0; i < size; i++)
                        L.add(false);
        }

        public void setSensor(int n, boolean b) {
                L.set(n, b);
        }
}
```



The Java library class Integer also performs caching. It creates a cache of 256 objects, for the integers between -128 and 127. Its valueOf method returns a reference to one of these constants if its argument is within that range; otherwise it creates a new object and returns it.

For example, consider the code of Listing 5-6. The first two calls to valueOf will return a reference to the cached Integer object for the value 127. The third and fourth calls will each create a new Integer object for the value 128. In other words, the code creates two new Integer objects, both having the value 128.

Listing 5-6. An Example of Integer Caching

```
List<Integer> L = new ArrayList<>();
L.add(Integer.valueOf(127)); // uses cached object
L.add(Integer.valueOf(127)); // uses cached object
L.add(Integer.valueOf(128)); // creates new object
L.add(Integer.valueOf(128)); // creates new object
```

Singleton Classes

A singleton class is a class that has a fixed number of objects, created when the class is loaded. It does not have a public constructor, so no additional objects can be created. It is called “singleton” because the most common situation is a class having a single instance.

```
public enum Boolean {
        TRUE(true), FALSE(false);

        private boolean value;
        private Boolean (boolean b) {value = b;}

        private boolean booleanValue() {
                return value;
        }

        public static Boolean valueOf (boolean b) {
                return (b ? TRUE : FALSE);
        }
}
```

Note that the syntactic differences are incredibly minor. The main difference concerns the definitions of the constants TRUE and FALSE, which omit both the declaration of their type and their call to the Boolean constructor. The values inside the parentheses denote the arguments to the constructor.

Beginners are often unaware of the correspondence between enums and classes because an enum is typically introduced as a named set of constants. For example, the following enum defines the three constants Speed.SLOW, Speed.MEDIUM, and Speed.FAST: `public enum Speed {SLOW, MEDIUM, FAST};`

This enum is equivalent to the class bellow. Note that each Speed constant is a reference to a Speed object having no functionality of interest.

```
public class Speed {
        public static final Speed SLOW = new Speed();
        public static final Speed MEDIUM = new Speed();
        public static final Speed FAST = new Speed();

        private Speed() { }
}
```

Because the constants in an enum are objects, they inherit the equals, toString, and other methods of Object. In the simple case of the Speed enum, its objects can do nothing else. The elegant thing about the Java enum syntax is that enum constants can be given as much additional functionality as desired. The default implementation of an enum's toString method is to return the name of the constant. For example, the following statement assigns the string "SLOW" to variable s.
`String s = Speed.SLOW.toString();`

Suppose instead that you want the Speed constants to display as musical tempos. Then you could override the toString method.

```
public enum Speed {
        SLOW("largo"), MEDIUM("moderato"), FAST("presto");
        private String name;

        private Speed(String name) {
                this.name = name;
        }

        public String toString() {
                return name;
        }
}
```

Singleton Strategy Classes

The OwnerStrategy interface has two implementing classes, Domestic and Foreign. Both of these classes have empty constructors and their objects are immutable. Consequently, all Domestic objects can be used interchangeably, as can all Foreign objects. Instead of creating new Domestic and Foreign objects on demand (which is what the AbstractBankAccount class currently does), it would be better for the classes to be singletons.

```
public enum Foreign implements OwnerStrategy {
        INSTANCE;

        public boolean isForeign() {
                return true;
        }
```

```java
	public int fee() {
		return 500;
	}
}
```

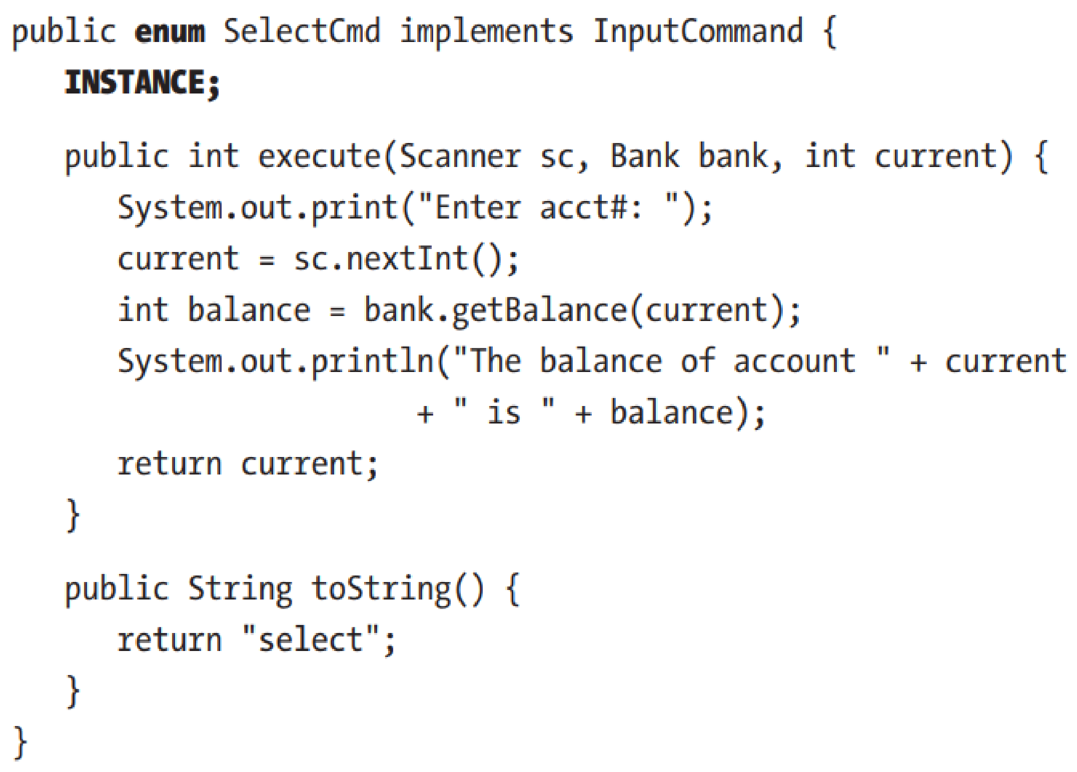
```java
public class AbstractBankAccount implements BankAccount {
	protected int acctnum;
	protected int balance = 0;
	protected OwnerStrategy owner = Domestic.INSTANCE;
	// ...
	public void setForeign(boolean b) {
		owner = b ? Foreign.INSTANCE : Domestic.INSTANCE;
	}
}
```

Or:

```java
public enum Owners implements OwnerStrategy {
	DOMESTIC(false,0,"domestic"), FOREIGN(true,500,"foreign");
	// ...
}
```

Commands

```java
private InputCommand[] commands = {
	QuitCmd.INSTANCE,
	NewCmd.INSTANCE,
	SelectCmd.INSTANCE,
	DepositCmd.INSTANCE,
	LoanCmd.INSTANCE,
	ShowCmd.INSTANCE,
	InterestCmd.INSTANCE,
	SetForeignCmd.INSTANCE
};
```



```
public enum SelectCmd implements InputCommand {
   INSTANCE;

   public int execute(Scanner sc, Bank bank, int current) {
      System.out.print("Enter acct#: ");
      current = sc.nextInt();
      int balance = bank.getBalance(current);
      System.out.println("The balance of account " + current
                          + " is " + balance);
      return current;
   }

   public String toString() {
      return "select";
   }
}
```

An alternative to having a separate enum for each command is to create a single enum containing all the commands.

```
public enum InputCommands implements InputCommand {
        SELECT("select", (sc, bank, current) ->{
                System.out.print("Enter account#: ");
                current = sc.nextInt();
                int balance = bank.getBalance(current);
                System.out.println("The balance of account " + current + " is " + balance);
                return current;
        }),
        // ...
}
```

Static Factory Methods

A factory method is a method whose job is to create objects. It encapsulates the details of object creation, and can hide the class of a newly constructed object. It can even hide the fact that it is returning a previously-created object instead of a new one.

The Java library contains many other static factory methods. One example is the static method asList in the class Arrays. The argument to this method is an array of object references and its return value is a list containing those references. The following code illustrates its use.

```
String[] names = {"joe", "sue", "max"};
List<String> L = Arrays.asList(names);
```

The asList method returns a list containing the elements of the supplied array, but it gives no other details. The method not only hides the algorithm for creating the list, it also hides the class of the list. This encapsulation gives the factory method considerable flexibility in how it chooses to create the list. For example, one option is for the method to create a new ArrayList object and then add each element of the array into it. But other options are possible.

```
public interface AccountFactory {
   static BankAccount createSavings(int acctnum) {
      return new SavingsAccount(acctnum);
   }

   static BankAccount createRegularChecking(int acctnum) {
      return new RegularChecking(acctnum);
   }

   static BankAccount createInterestChecking(int acctnum) {
      return new InterestChecking(acctnum);
   }

   static BankAccount createAccount(int type, int acctnum) {
      BankAccount ba;
      if (type == 1)
         ba = createSavings(acctnum);
      else if (type == 2)
         ba = createRegularChecking(acctnum);
```

The first three methods hide the subclass constructors. The createAccount method encapsulates the decision about which account type has which type number. This decision had previously been made by Bank (as was shown in Listing 5-19) as well as SavedBankInfo (see Listing 3-17). By moving the decision to AccountFactory, those classes can now call createAccount without needing to know anything about how account types are implemented.

For example, Listing 5-21 shows the version 12 newAccount method of Bank, modified to call the createAccount method. The SavedBankInfo class is modified similarly but is not shown here.

Listing 5-21. The Version 12 newAccount Method of Bank

```
public int newAccount(int type, boolean isforeign) {
   int acctnum = nextacct++;
   BankAccount ba =
               AccountFactory.createAccount(type, acctnum);
   ba.setForeign(isforeign);
   accounts.put(acctnum, ba);
   return acctnum;
}
```

Factory Objects

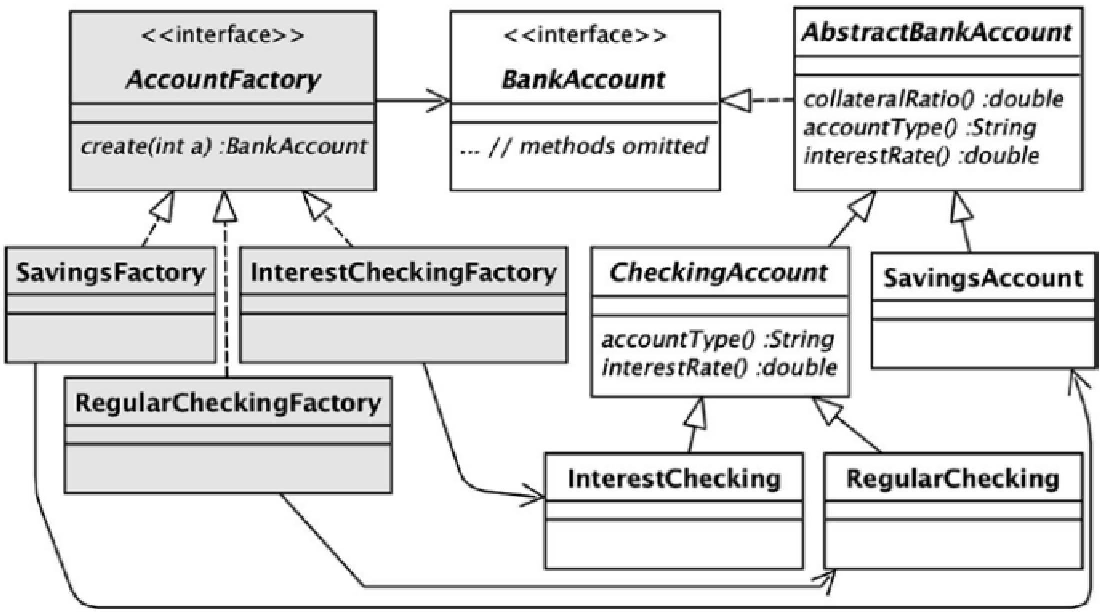
Recall that in the command pattern, each command is an object. To execute a command you first obtain the desired command object and then call its execute method. Analogously, to execute a factory command you first obtain the desired factory object and then call its create method. The following code illustrates how these two steps combine to create a new BankAccount object from a factory object.

The AccountFactory class greatly improves the banking demo, because the demo now has a single place to hold its knowledge about the BankAccount subclasses. Of course, AccountFactory is coupled to every BankAccount subclass, which implies that any changes to the subclasses will require a modification to AccountFactory—thereby violating the Open/Closed rule. But at least this violation has been limited to a single, well-known place. It is possible to improve on this design. The idea is that a static factory method is essentially a command to create an object. If you have several related static factory methods (as AccountFactory does) then you can create a more object-oriented design by using the command pattern from Java Program Design - Anotações Ch. 4

```java
AccountFactory af = new SavingsFactory();
BankAccount ba = af.create(123);
```

```java
public class SavingsFactory implements AccountFactory {
	public BankAccount create (int acctnum) {
		return new SavingsAccount (acctnum);
	}
}
```

```java
public interface AccountFactory {
	BankAccount create (int acctnum);

	static BankAccount createAccount (int type, int acctnum) {
		AccountFactory af;
		if (type == 1)
			af = new SavingsFactory();
		else if (type == 2)
			af = new RegularCheckingFactory();
		else
			af = new InterestCheckingFactory();
	}
}

public interface BankAccount extends Comparable<BankAccount> {

	// ...

	static BankAccount createSavingsWithDeposit(
		int acctnum, int n
	) {
		AccountFactory af = new SavingsFactory();
		BankAccount ba = af.create(acctnum);
		ba.deposit(n);
		return ba;
	}

}
```



Cached Factory Objects

The code above should help solidify your understanding of how factories work—namely that the creation of an object requires two steps: creating a factory object, and calling its create method. The code may also leave you with the question of why anyone would want to do things this way. What is the advantage of using factory objects? The answer has to do with the fact that factory objects do not need to be created at the same time as the objects they create. In fact, it usually makes sense to create the factory objects early and cache them.

The answer has to do with the fact that factory objects do not need to be created at the same time as the objects they create. In fact, it usually makes sense to create the factory objects early and cache them.

```
public interface AccountFactory {
        BankAccount create(int acctnum);

        static AccountFactory[] factories = {
                new SavingsFactory(),
                new RegularCheckingFactory(),
                new InterestCheckingFactory()
        }

        static BankAccount createAccount (int type, int acctnum) {
                AccountFactory af = factories[type-1];
                return af.create(acctnum);
        }
}
```

Note the implementation of the createAccount method. It no longer needs to use an if-statement to choose which type of account to create. Instead, it can simply index into the precomputed array of factory objects. This is a big breakthrough in the design of AccountFactory. Not only does it eliminate the annoying if-statement but it also brings the interface very close to satisfying the Open/Closed rule. To add a new account factory, you now only need to create a new factory class and add an entry for that class into the factories array.

Of course, instead of caching the factory objects manually, it would be better to implement them as enum constants. The constructor has two arguments: a string indicating the display value of the constant, and a lambda expression giving the code for the create method.

```
public enum AccountFactories implements AccountFactory {
        SAVINGS("Savings",
                acctnum -> new SavingsAccount(acctnum)),
        REGULAR_CHECKING("Regular checking",
                acctnum -> new RegularChecking(acctnum)),
        INTEREST_CHECKING("Interest checking",
                acctnum -> new InterestChecking(acctnum));

        private String name;
        private AccountFactory af;

        private AccountFactories(String name, AccountFactory af) {
                this.name = name;
                this.af = af;
        }

        // Each constant has a name and an `AccountFactory` implementation that creates a specific type
of `BankAccount` based on the account number.

        public BankAccount create(int acctnum) {
                return af.create(acctnum);
        }

        public String toString() {
                return name;
        }
}
```

```
public interface AccountFactory {
        BankAccount create (int acctnum);
        static AccountFactory[] factories = AccountFactories.values();

        static BankAccount createAccount (int type, int acctnum) {
                AccountFactory af = factories[type-1];
                return af.create(acctnum);
        }
}
```

```
public interface BankAccount extends Comparable<BankAccount> {
        // ...
        static BankAccount createSavingsWithDeposit (
                int acctnum, int n
        ) {
                AccountFactory af = AccountFactory.SAVINGS;
                BankAccount ba = af.create(acctnum);
                ba.deposit(n);
                return ba;
        }
}
```

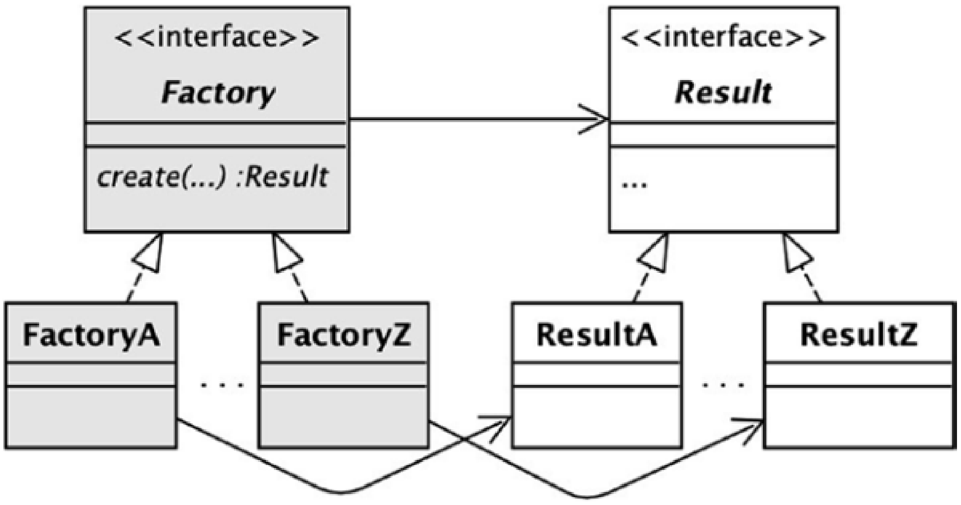
One final point: You might recall that the constant NEW in the version 13 InputCommands enum asks the user to choose from a list of account types. How can you ensure that the type numbers presented to the user stay in synch with the type numbers associated with the AccountFactory array? The solution is to modify NEW so that it constructs the user message based on the contents of the AccountFactories.values array.

```
private static String message;

static {
   AccountFactory[] factories = AccountFactories.values();
   message = "Enter Account Type (";
   for (int i=0; i<factories.length-1; i++)
      message += (i+1) + "=" + factories[i] + ", ";
   message += factories.length + "="
           + factories[factories.length-1] +")";
}

private static void printMessage() {
   System.out.print(message);
}
```

The Factory Pattern

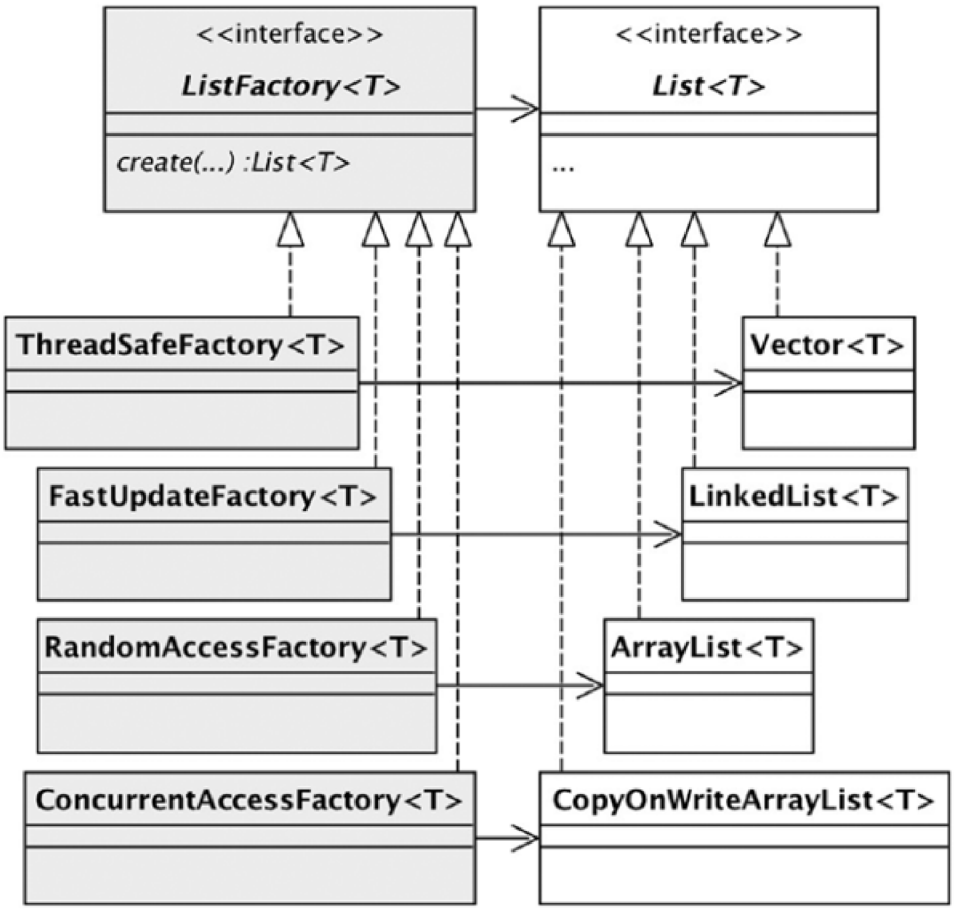
The class diagram illustrates a typical use of factory classes, namely that the classes in the factory hierarchy objects belonging to a second, parallel hierarchy called the result hierarchy. Each class in the factory hierarchy has a corresponding class in the result hierarchy.

This design is sufficiently common that it has a name: the factory pattern.



Typically, the need for the factory pattern arises when you have a result hierarchy and you want clients to be able to create result objects without knowing the names of each result subclass. The factory pattern says that you should create a parallel factory hierarchy so that your clients can create a result object by calling the create method of the appropriate factory object. For an example, consider the List interface. The Java library has several classes that implement List, with each class having a different purpose. For example, Vector is thread-safe; CopyOnWriteArrayList enables safe concurrent access; ArrayList is random-access; and LinkedList supports fast inserts and deletes. Suppose that you want your clients to be able to create List objects based on these characteristics, but you don't want them to choose the classes themselves. **You might have several reasons for this decision: perhaps you don't want your clients to have to know the name of each class and its characteristics, or you want clients to choose from only these four classes, or you want the flexibility to change the class associated with a given characteristic as time goes on.**

Your solution is to use the factory pattern. You create an interface ListFactory, whose factory classes are ThreadSafeFactory, ConcurrentAccessFactory, RandomAccessFactory, and FastUpdateFactory. Each factory creates an object from its associated result class.



Factories for Customized Objects

Another way to use a factory hierarchy is to have the factory classes create objects from the same result class. In this case, the purpose of each factory is to customize its result object in a particular way. This section examines three examples of this design technique.

```
public enum AccountFactories implements AccountFactory {
   SAVINGS("Savings",
         acctnum -> new SavingsAccount(acctnum)),
   REGULAR_CHECKING("Regular checking",
         acctnum -> new RegularChecking(acctnum)),
   INTEREST_CHECKING("Interest checking",
         acctnum -> new InterestChecking(acctnum)),
   NEW_CUSTOMER("New Customer Savings",
         acctnum -> {
            BankAccount result = new SavingsAccount(acctnum);
            result.deposit(1000); // $10 for free!
            return result; });
   ...
}
```