Java Program Design - Anotações Ch. 2

Data: 20/02/2023
Tags: #SoftwareEngineering #java #POO
PDF: Java Program Design Principles, Polymorphism, and Patterns (Edward Sciore).pdf

---

Polymorphism

Each **class** in a well-designed program denotes a distinct concept with its **own set of responsibilities**. Nevertheless, it is possible for **two (or more) classes to share some common functionality**.

> For example, the Java classes HashMap and TreeMap are distinct implementations of the concept of a “map,” and both support the methods get, put, keySet, and so on. The ability of a program to take advantage of this commonality is known as polymorphism.

It can be argued that polymorphism is the most important design concept in object-oriented programming. A solid understanding of polymorphism (and interfaces) is critical for any good Java programmer.

The Need for Polymorphism

Suppose you have been asked to modify version 4 of the banking demo to support two kinds of bank account: savings accounts and checking accoounts. Savings accounts correspond to the bank accounts of version 4.
Checking accounts differ from savings in the following three ways:

- When authorizing a loan, a checking account needs a balance of two thirds the loan amount, whereas savings accounts require only one half the loan amount.
- The bank gives periodic interest to savings accounts but **not** checking accounts
- The `toString` method of an account will return "Savings Account" or "Checking Account", as appropriate.

One straightforward (and somewhat naive) way to implement these two types of accounts would be to modify the code for `BankAccount`.
For example, `BankAccount` could have a variable that holds the type of the account: a value of 1 denotes a savings account and a value of 2 denotes a checking account. The methods `hasEnoughCollateral`, `toString`, and `addInterest` would use an if-statement to determine which account type to handle.

```java
private int type;

public BankAccount(int acctnum, int type) {
   this.acctnum = acctnum;
   this.type = type;
}
...
public boolean hasEnoughCollateral(int loanamt) {
   if (type == 1)
      return balance >= loanamt / 2;
   else
      return balance >= 2 * loanamt / 3;
}
```

```
    public String toString() {
       String typename = (type == 1) ?
                              "Savings" : "Checking";
       return typename + " Account " + acctnum
                 + ": balance=" + balance + ", is "
                 + (isforeign ? "foreign" : "domestic");
    }

    public void addInterest() {
       if (type == 1)
          balance += (int)(balance * rate);
    }
 }
```

Although this code is a straightforward modification to BankAccount, it has two significant **problems**.

First, **the if-statements are inefficient**. Every time one of the revised methods gets called, it must go through the conditions in its if-statement to determine what code to execute. Moreover, increasing the number of account types will cause these methods to execute more and more slowly.

Second (and more importantly), the code is difficult to modify and thus violates the fundamental design principle ("A program should be design so that any change to it will affect only a small, predictable portion of the code.")
Each time that another account type is added, another condition must be added to each if-statement. This is tedious, time-consuming, and error prone. If you forget to update one of the methods correctly, the resulting bug might be difficult to find.

The way to avoid this if-statement problem is to **use a separate class for each type of account**. Call these classes SavingsAccount and CheckingAccount. The advantage is that each class can have its own implementation of the methods, so there is no need for an if-statement. Moreover, whenever you need to add another type of bank account you can simply create a new class.

But how then does the Bank class deal with **multiple account classes**? You don't want the bank to hold a separate **map for each type of account** because that will just introduce other modifiability problems. For example, suppose that there are several account types **that give interest**, with each account type having its own map. The code for the addInterest method **will need to loop through each map individually**, which means that each new account type will require you to add a new loop to the method.

The only good solution is for all account objects, regardless of their class, to be in a single map. Such a map is called *polymorphic*. Java uses interfaces to implement polymorphism. The idea is for version 5 of the banking demo to replace the `BankAccount` class with a `BankAccount` interface, i.e., the account map will still be defined like this:

```
private HashMap<Integer, Integer> accounts;
```

However, `BankAccount` is now an interface, whose objects can be from either `SavingsAccount` or `CheckingAccount`. The next section explains how to implement such a polymorphic map in Java.

Interfaces

A Java *interface* is primarily a named set of method headers. (Interfaces also have other features that will be discussed later.) An interface is similar to the API of a class. The difference is that a class's API is inferred from its public methods, whereas an interface specifies the API explicitly without providing any code:

> An interface provides a way to define a **contract** for behavior that a class must implement without specifying how it should be implemented.

```
public interface BankAccount {
        public abstract int getAccNum();
        public abstract int getBalance();
        public abstract boolean isForeign();
        public abstract void setForeign(boolean isforeign);
        public abstract void deposit(int amt);
        public abstract boolean hasEnoughCollateral(int loanamt);
        public abstract String toString();
}
```

The keyword `abstract` indicates that the method declaration contains only the method's header and tells the compiler that its code will be specified elsewhere.
The `abstract` and `public` keywords are optional in an interface declaration because interface methods are public and abstract by default.

> e.g.

```
public interface MyInterface {
    void myMethod();
    int myOtherMethod(String arg);
}
```

> In this example, the "MyInterface" interface declares two abstract methods, "myMethod" and "myOtherMethod". Any class that implements the "MyInterface" interface must provide concrete implementations for these methods.
>
> In addition to abstract methods, an interface can also contain constants and default methods (methods with an implementation). Interfaces are commonly used in Java to **define contracts for classes that implement a certain behavior or functionality**.

The code for an interface's methods is suplied by classes that *implement* the interface. Suppose that `I` is an interface. A class indicates its intent to implement `I` by adding the clause `implements I` to its header.
If a class implements an interface then it is obligated to implement all methods declared by that interface. The compiler will generate an error if the class does not contain those methods.

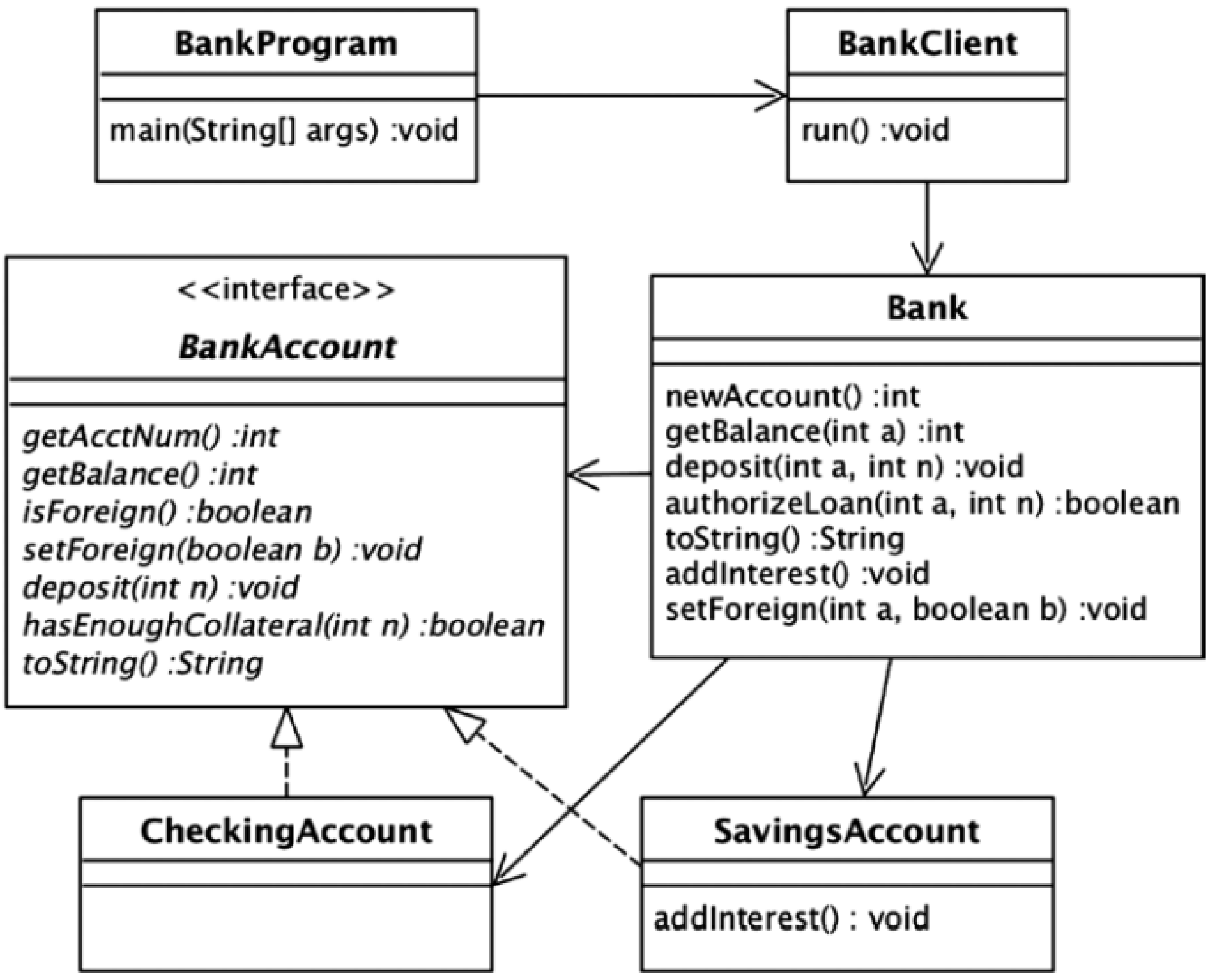
So, in version 5 of the banking demo, the classes `CheckingAccount` and `SavingsAccount` both implement the `BankAccount` interface.

```
public class SavingsAccount implements BankAccount {
   private double rate = 0.01;
   private int acctnum;
   private int balance = 0;
   private boolean isforeign = false;

   public SavingsAccount(int acctnum) {
      this.acctnum = acctnum;
   }
   ...
   public boolean hasEnoughCollateral(int loanamt) {
      return balance >= loanamt / 2;
   }

   public String toString() {
      return "Savings account " + acctnum
             + ": balance=" + balance
             + ", is " + (isforeign ? "foreign" : "domestic");
   }

   public void addInterest() {
      balance += (int) (balance * rate);
   }
}

public class CheckingAccount implements BankAccount {
   // the rate variable is omitted
   private int acctnum;
   private int balance = 0;
   private boolean isforeign = false;
```

```
    public CheckingAccount(int acctnum) {
       this.acctnum = acctnum;
    }
    ...
    public boolean hasEnoughCollateral(int loanamt) {
       return balance >= 2 * loanamt / 3;
    }

    public String toString() {
       return "Checking account " + acctnum + ": balance="
                   + balance + ", is "
                   + (isforeign ? "foreign" : "domestic");
    }

    // the addInterest method is omitted
 }
```

In general, a class may implement any number of interfaces. Its only obligation is to have code for each method of each interface that it implements. A class is also free to have methods in **addition to the ones required by its implemented interfaces**. For example, `SavingsAccount` has the public method `addInterest`, which is not part of its `BankAccount` interface.

An interface is denoted in a class diagram by a rectangle, similarly to a class. The name of the interface goes in the rectangle. To distinguish an interface from a class, the annotation `<<interface>>` appears above its name. The interface name and its methods are italicized to emphasize that they are abstract.



> Bank has dependencies to `BankAccount`, `SavingsAccount` and `CheckingAccount` because its code uses variables of all three types, as will be seen in the next section.

Reference Types

This section examines **how interfaces affect the typing of variables** in a Java program. Each Java variable has a declared type, and this type determines the kind of value the variable can hold.

If a variable holds a **basic** value (such as an int or a float) then its type is said to be a *primitive type*. If the variable holds an **object** reference then its type is said to be a *reference type*.

Each class and each interface defines a reference type. If a variable is class-typed then it can hold a reference to any object of that class. If a variable is interface-typed then it **can hold a reference to any object whose class implements that interface**. For example, consider the following two statements:

```
SavingsAccount sa = new SavingsAccount(1);
BankAccount    ba = new SavingsAccount(2);
```

The first statement stores a `SavingsAccount` reference in the class-typed variable `sa`. This statement is legal because the class of the object reference is the same as the type of the variable.
The second statement stores a `SavingsAccount` reference in the interface-typed variable `ba`. This statement is **also legal** because of the object reference implements the type of the variable

> BankAccount (an interface => *interface-typed*) can hold reference to an object whose class implements the interface, i.e., SavingsAccount.

The type of a variable determines which methods the program can invoke on it. **A class-typed variable can call only the public methods of that class. An interface-typed variable can call only the methods defined by that interface.** Continuing the preceding example, consider these four statements:

```
sa.deposit(100);
sa.addInterest();
ba.deposit(100);
ba.addInterest(); // Illegal! The public method addInterest isn't defined in the interface definition,
only in the class SavingsAccount that implements that interface
```

This example points out that storing an object reference in an interface-typed variable can “cripple” it. Variables sa and ba both hold similar SavingsAccount references. However, sa can call addInterest whereas ba cannot

So **what is the point of having interface-typed variables**? The primary advantage of an interface-typed variable is that it can hold references to objects from different classes. For example, consider the following code:

```
BankAccount ba = new SavingsAccount(1);
ba = new CheckingAccount(2);
```

In the first statement, variable ba holds a SavingsAccount reference. In the second statement, it holds a CheckingAccount reference. **Both statements are legal because both classes implement BankAccount.** This feature is especially useful when a variable can hold more than one element. For example, consider the following statements.

```
BankAccount[] accts = new BankAccount[2];
accts[0] = new SavingsAccount(1);
accts[1] = new CheckingAccount(2);
```

The variable accts is an array whose elements have the type BankAccount. It is **polymorphic** because it can store object references from SavingsAccount and CheckingAccount. For example, the following loop deposits 100 into every account of the accts array, regardless of its type.

```
for (int i = 0; i < accts.length; i++) {
        accts[i].deposit(100);
}
```

```
public int newAccount(int type, boolean isforeign) {
   int acctnum = nextacct++;
   BankAccount ba;
   if (type == 1)
      ba = new SavingsAccount(acctnum);
   else
      ba = new CheckingAccount(acctnum);
   ba.setForeign(isforeign);
   accounts.put(acctnum, ba);
   return acctnum;
}

public int getBalance(int acctnum) {
   BankAccount ba = accounts.get(acctnum);
   return ba.getBalance();
}
...
public void addInterest() {
   for (BankAccount ba : accounts.values())
      if (ba instanceof SavingsAccount) {
         SavingsAccount sa = (SavingsAccount) ba;
         sa.addInterest();
      }
}
```

Consider the method newAccount. It now has an additional parameter, which is an integer denoting the account type. The value 1 denotes a savings account and 2 denotes a checking account. The method creates an object of the specified class and stores a reference to it in the variable ba. Because this variable has the type BankAccount, it can hold a reference to either a SavingsAccount or a CheckingAccount object. Consequently, both savings and checking accounts can be stored in the accounts map.

Now consider the method getBalance. Since its variable ba is interface-typed, the method does not know whether the account it gets from the map is a savings account or a checking account. But it doesn't need to know. The method simply calls ba.getBalance, which will execute the code of whichever object ba refers to. The omitted methods are similarly polymorphic. The method addInterest is more complex than the other methods. An understanding of this method requires knowing about type safety, which will be discussed next.

Type Safety

The compiler is responsible for assuring that each variable holds a value of the proper type. We say that the compiler assures that the program is *type-safe*. **If the compiler cannot guarantee that a value has the proper type then it will refuse to compile that statement.** For example, consider the following code:

```
SavingsAccount sa1 = new SavingsAccount(1);
BankAccount ba1 = new CheckingAccount(2);
BankAccount ba2 = sa1;
BankAccount ba3 = new Bank(...); // Unsafe
SavingsAccount sa2 = ba2;        // Unsafe
```

The first statement stores a reference to a SavingsAccount object in a SavingsAccount variable. This is clearly type-safe. The second statement stores a reference to a CheckingAccount object in a BankAccount variable. This is type-safe because CheckingAccount implements BankAccount. The third statement stores the reference held by sa1 into a BankAccount variable. Since sa1 has the type SavingsAccount, the compiler can infer that its reference must be to a SavingsAccount object and thus can be safely stored in ba2 (because SavingsAccount implements BankAccount).

The fourth statement is clearly not type-safe because Bank does not implement BankAccount. The variable ba2 in the fifth statement has the type BankAccount, so the compiler infers that its object reference could be from **either SavingsAccount or CheckingAccount. Since a CheckingAccount reference cannot be stored in a SavingsAccount variable, the statement is not type-safe.** The fact that ba2 actually holds a reference to a SavingsAccount object is *irrelevant*.

Type Casting

The compiler is very conservative in its decisions. If there is any chance whatsoever that a variable can hold a value of the wrong type then it will generate a compiler error. For example, consider the following code:

```
BankAccount ba = new SavingsAccount(1);
SavingsAccount sa = ba;
```

It should be clear that the second statement is type-safe because the two statements, taken together, imply that variable `sa` will hold a SavingsAccount reference when compiling the second one. It knows only that variable `ba` is of the type `BankAccount` and thus could be holding a `CheckingAccount` value.
It therefore generates a compiler error.

In cases such as this, you can use a *typecast* to overrule the compiler. For example, the preceding code can be rewritten as follows:

```
BankAccount ba = new SavingsAccount(1);
SavingsAccount sa = (SavingsAccount) ba;
```

The typecast assures the compiler that the code really is type-safe, and that you assume all responsability for any incorrect behaviour. The compiler then obeys your request and compiles the statement. If you are wrong then the program will throw a `ClassCastException` at runtime.

It is now possible to consider the `addInterest` method. This method iterates through all the accounts but adds interest only to the savings accounts. Since the elements of the variable `accounts` are of type `BankAccount`, and `BankAccount` does not have an `addInterest` method, some fancy footwork is needed to ensure type-safety.

```
public void addInterest() {
        for (BankAccount ba : account.values())
                if (ba instanceof SavingsAccount) {
                        SavingsAccount sa = (SavingsAccount) ba;
                        sa.addInterest();
                }
}
```

(end of version 5)

Transparency

The technique of combining a call of `instanceof` with a typecast gives correct results, but it violates the fundamental design principle. The problem is that the code specifically mentions class names. If the bank adds another account type that also gives interest (shuch as a "money market account") then you would need to modify the `addInterest` method to deal with it.

The if-statement problem is rearing its ugly head again. Each time a new kind of account is created, you will need to examine every relevant portion of the program to decide wether that new class needs to be added to the if-statement. For large programs, this is a daunting task that has potencial for creation of many bugs.

The way to eliminate these problems is to add the `addInterest` method to the `BankAccount` interface. Then the `addInterest` method of `Bank` could call `addInterest` on each of its accounts without caring which class of an object they belonged to. Such a design is called *transparent* because the class of an object reference is invisible to a client. We express these ideas in the following rule:

The Rule of Transparency

A client should be able to use an interface without needing to know the classes that implement that interface.

This transparency requires changes to the code for `BankAccount`, `CheckingAccount` and `Bank`. The `BankAccount` interface needs an additional method header for `addInterest`:

```
public interface BankAccount {
        // ...
        void addInterest();
}
```

`CheckingAccount` must implement the additional method `addInterest`, which simply needs to do nothing.

```
public class CheckingAccount implements BankAccount {
        // ...
        public void addInterest() {
                // do nothing
        }
}
```

And `Bank` has a new, transparent implementation of `addInterest`:

```
public class Bank {
        // ...
        public void addInterest() {
                for (BankAccount ba : accounts.values()) {
                        ba.addInterest();
                }
        }
}
```

The Open-Closed Rule

Now, consider how you would have to change the version 6 banking demo:

- You would write a new class `MoneyMarketAccount` that implemented the `BankAccount` interface
- You would modify the `newAccount` method of `BankClient` to display a different message to the user, indicating the account type for `MoneyMarketAccount`.

- You would modify the `newAccount` method in `Bank` to create new `MoneyMarketAccount` objects.

=> *Modification* - existing classes change;
=> *Extension* - new classes are written;

In general, modification is the one that tends to be the source of bugs. This insight leads to the following rule:

The Open/Closed Rule

To the extent possible, a program should be open for extension
but closed for modification.

(The first task can be done by extension, and the other two by small modifications - the techniques of chapter 5 will make it possible to further reduce the modifications required for these two tasks.)

The Comparable Interface

Suppose that the bank has asked you to modify the banking demo so that bank accounts can be compared according to their balanced. That is, it wants `ba1 > ba2 if b1 has more money than ba2`.

The java library has an interface especially for this purpose, called `Comparable<T>`.

```
public interface Comparable<T> {
        int compareTo(T t);
}
```

The call `x.compareTo(y)` returns a number greater than 0 if `x > y`, a value less than 0 `if x < y`, and 0 if `x = y`. Many classes from the java library are comparable.
For example:

```
String s1  = "abc";
String s2  = "x";
int result = s1.compareTo(s2);
```

Version 6 of the banking demo modifies classes `SavingsAccount` and `CheckingAccount` to implement `Comparable<BankAccount>`. Each class now has a compareTo method and its header declares that it implements `Comparable<BankAccount>`. Listing gives the relevant code for `SavingsAccount`. The code for `CheckingAccount` is similar.

```
public class SavingsAccount implements BankAccount,
                                                   Comparable<BankAccount> {
        // ...
        public int compareTo (BankAccount ba) {
                int bal1 = getBalance();
                int bal2 = ba.getBalance();
                if (bal1 == bal2)
                        return getAcctNum() - ba.getAcctNum();
                else
                        return bal1 - bal2;
        }
}
```

Subtypes

Although the version 6 code declares both `SavingsAccount` and `CheckingAccount` to be comparable, that is not the same as requiring that all bank accounts be comparable. This is a serious problem. For example, consider the following statements. The compiler will refuse to compile the third statement because `BankAccount` variables are not required to have a `compareTo` method.

```
BankAccount ba1 = new SavingsAccount(1);
BankAccount ba2 = new SavingsAccount(2);
int a = ba1.compareTo(ba2); // unsafe!
```

This problem also occurs in the class `CompareBankAccounts`.

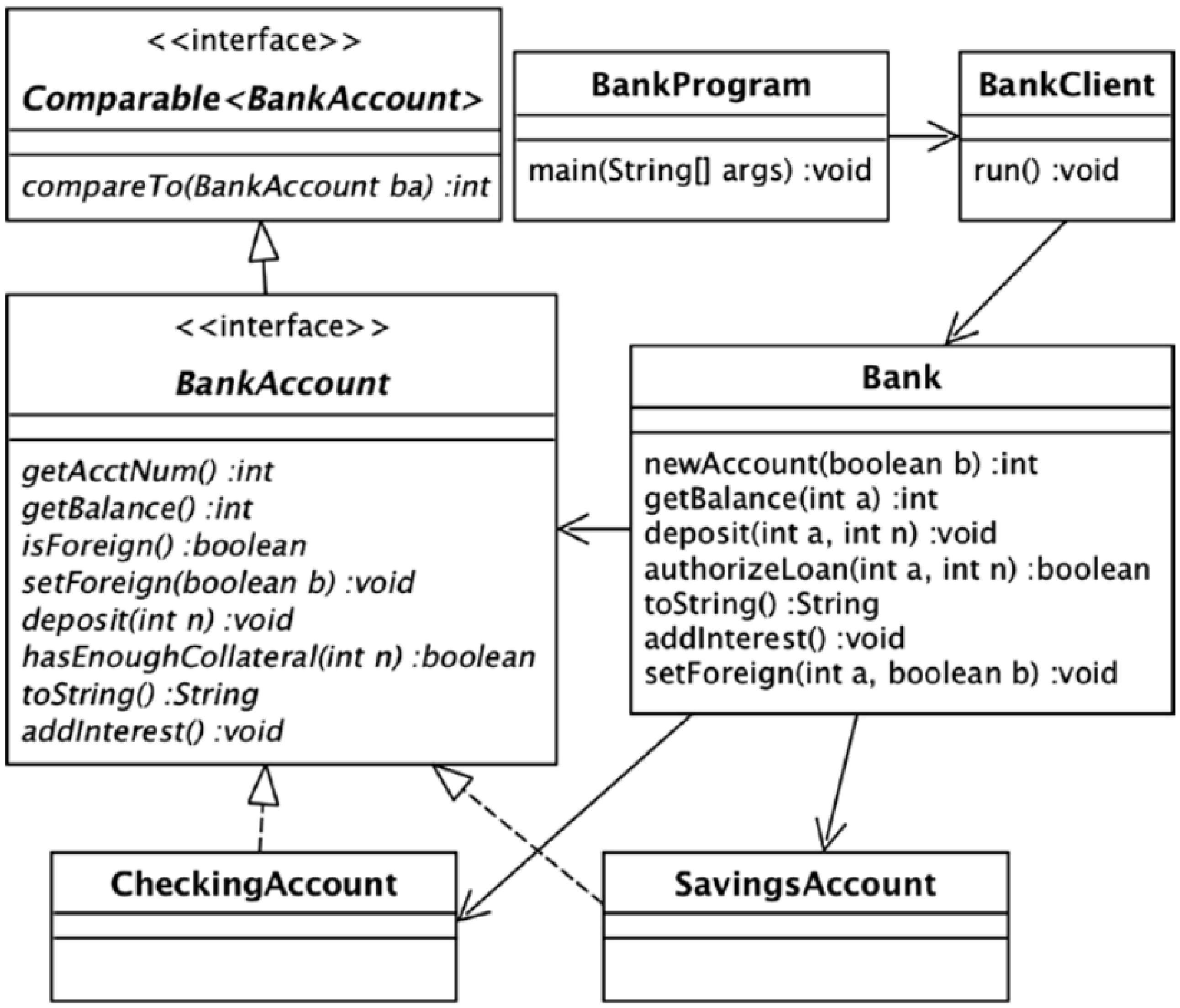
```
public class CompareBankAccounts {
        public static void main(String[] args) {
                ArrayList accts = initAccts();
                BankAccount maxacct1 = findMax(accts);
                BankAccount maxacct2 = Collections.max(accts);
                ...
        }

        private static BankAccount findMax(ArrayList a) {
                BankAccount max = a.get(0);
                for (int i = 1; i < a.size(); i++) {
                        if (a.get(i).compareTo(max) > 0)
                                max = a.get(i); } return max;
                }
                return max;
}
```

The solution to both examples is to assert that all classes that implement `BankAccount` also implement `Comparable<BankAccount>`.
Formally speaking, we say that `BankAccount` needs to be a *subtype* of `Comparable<BankAccount>`. You specify subtypes in Java by using the keyword extends.

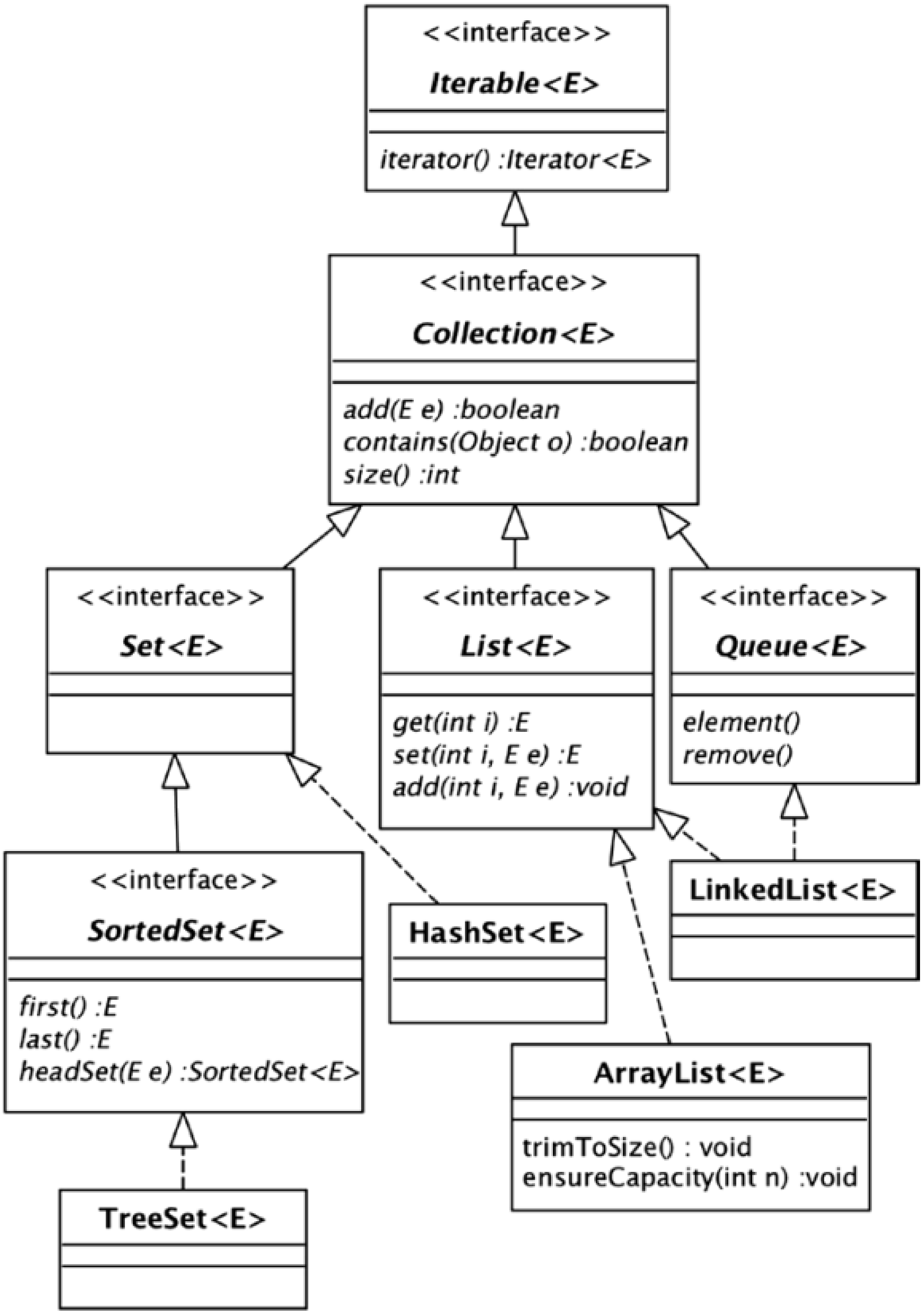
```
public interface BankAccount extends Comparable<BankAccount> {
        // ...
}
```

Important

The `extends` keyword indicates that if a class implements `BankAccount` then it must also implement `Comparable<BankAccount>`. Consequently, the classes `SavingsAccount` and `CheckingAccount` no longer need to explicitly implement `Comparable<BankAccount>`. With this change, `CompareBankAccounts` compiles and executes correctly.



The Java Collection Library



The Liskov Substitution Principle

If type X extends type Y then an object of type X can always be used wherever an object of type Y is expected.

> For example, consider the fact that List extends Collection. The LSP implies that List objects can substitute for Collection objects. In other words, if someone asks you for a collection then you can reasonably give them a list because the list has all the methods it needs to behave as a collection.

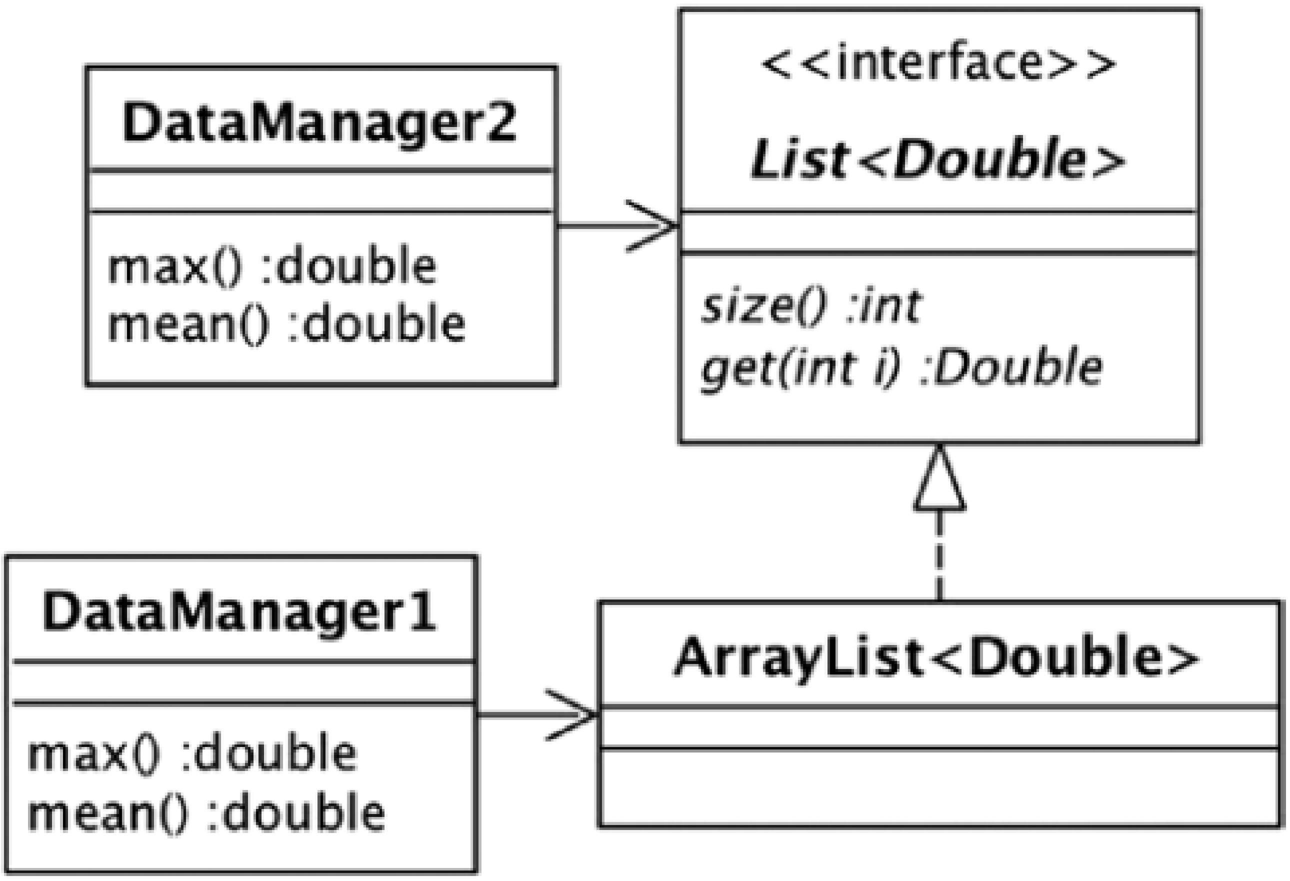
The Rule of Abstraction

```
public class DataManager1 {

        private ArrayList<Double> data;

        public DataManager1 (ArrayList<Double> d) {
                data = d;
        }

        public double max() {
                return Collections.max(data);
        }

        public double mean() {
                double sum = 0.0;
                for (int i = 0; i < data.size(); i++) {
                        sum += data.get(i);
                }
                return sum / data.size();
        }

}
```

Although this class executes correctly, it is poorly designed. Its problem is that it works only for data stored in `ArrayList` objects. This restriction is unnecessary beacause there is nothing in the code that applies only to array lists.

```
public class DataManager2 {

        private List<Double> data;

        public DataManager2(Lists<Double> d) {
                data = d;
        }

        // ...
}
```

The code for `DataManager1` and `DataManager2` are identical, except for the two places where `ArrayList` has been replaced by `List`. These two classes and their dependencies can be expressed in the class diagram:



The added flexibility of class `DataManager2` results from the fact that it is dependent on the interface `List`, which is more abstract then the `DataManager1` dependency on `ArrayList`. This insight is true in general, and can be expressed as the following rule of Abstraction.

The Rule of Abstraction

A class's dependencies should be as abstract as possible.

Although DataManager2 is better than DataManager1, could it be made even better by changing its dependency on List to something more abstract, such as Collection? At first glance you would have to say "no" because the implementation of the mean method uses the List-based method get. If you want the class to work for any collection then you would need to write mean so that it uses only methods available to collections. Fortunately, such a rewrite is possible.

```
public class DataManager3 {
        private Collection<Double> data;
        public DataManager3 (Collection<Double> d) {
                data = d;
        }

        public double max() {
                return Collection.max(data);
        }

        public double mean() {
                double sum = 0.0;
                for (double d : data)
                        sum += d;
```

```java
                return sum / data.size();
        }
}
```

The rule of Abstraction can also be applied to the banking demo. Consider for example the dependency between Bank and HashMap. A Bank object has the variable accounts, which maps an account number to the corresponding BankAccount object. The type of the variable is HashMap. The rule of Abstraction suggests that the variable should have the type Map instead. That statement is changed in the version 6 code.

Adding Code to an Interface

At the beginning of this chapter the interface was defined to be a set of method headers, similar to an API. Under this definition, an interface cannot contain code. The Java 8 release loosened this restriction so that an interface can define methods, although it still cannot declare variables.
This section examines the consequences of this new ability.

```java
public interface BankAccount extends Comparable<BankAccount> {

        // ...

        static BankAccount createSavingsWithDeposit (int acctnum, int n) {
                BankAccount ba = new SavingsAccount(acctnum);
                ba.deposit(n);
                return ba;
        }

        default boolean isEmpty() {
                return getBalance() == 0;
        }

}
```

Both methods are examples of *convenience methods*. A convenience method does not introduce any new functionality. Instead, it leverages existing functionality for the convenience of clients. The method createSavingsWithDeposit creates a savings account having a specified initial balance. The method isEmpty returns true if the account's balance is zero, and false otherwise.

Interface methods are either *static* or *default*. A static method has the keyword *static*, which means the same as it does in a class. **A default method is nonstatic, wich indicates that an implementing class my override the code if it wishes**.
The idea is that the interface provides a generic implementation classes. But a specific class may be able to provide a better, more efficient implementation. For example, suppose that a money-market savings account requires a minimum balance of $100. Then it knows that the account will never be empty, and so it can overwrite the default `isEmpty` method to one that immediately returns false without having to examine the balance.

For a more interesting example of a default method consider the question of how to sort a list. The Java class `Collections` has the static method `sort`. You pass two arguments to the `sort` method - a list and a comparator - and it sorts the list for you. (A comparator is an object that specifies a sort order). Old way to sort a list:

```java
Scanner scanner = new Scanner(System.in);
List<String> words = new ArrayList<>();
for (int i = 0; i < 10; i++)
        words.add(scanner.next());
Collections.sort(words, null);
```

The problem with this sort method is that there is no good way to sort a list without knowing how it is implemented. The solution used by the `Collections` class is to copy the elements of the list into an array, sort the array, and then copy the sorted elements back to the list.

```java
public class Collections {
        // ...
        static <T> void sort(List<T>, Comparator<T> comp) {
                Object[] a = L.toArray();
                Array.sort(a, (Comparator) comp);
                for (int i = 0; i < L.size(); i++)
                        L.set(i, (T) a[i]);
        }
}
```

Although this code will work for any list, it has the overhead of copying the list elements to an array and back. This overhead is wasted time for some list implementations. An array list, for example, saves its list elements in an array, so it would be more efficient to sort that array directly. This situation means that a truly efficient list sorting method cannot be transparent. It would need to determine how the list is implemented and then use a sorting algorithm specific to that implementation.

Java 8 addresses this problem by making sort be a default method of the List interface. The code for List.sort is a refactoring of the Collections.sort code:

```java
public interface List<T> extends Collection<T> {
        // ...
        default void sort (Comparator<T> comp) {
                Object[] a = toArray();
                Arrays.sort(a, (Comparator) comp);
                for (int i = 0; i < size(); i++)
                        set(i, (T) a[i]);
        }
}
```

`Collections.sort(L, null);` <=> `L.sort(null);`

The second, and more important benefit is that lists can be handled transparently. The default implementation of sort works for all implementations of List. However, any particular List implementation (such as ArrayList) can choose to override this method with a more efficient implementation of its own.

Summary

Polymorphism is the ability of a program to leverage the common functionality of classes. Java uses interfaces to support polymorphism—the methods of an interface specify some common functionality, and classes that support those methods can choose to implement that interface. For example, suppose classes C1 and C2 implement interface I:

```java
public interface I {...}
public class C1 implements I {...}
public class C2 implements I {...}
```

A program can now declare variables of type I, which can then hold references to either C1 or C2 objects without caring which class they actually refer to.

This chapter examined the power of polymorphism and gave some basic examples of its use. It also introduced four design rules for using polymorphism appropriately:

- The rule of Transparency states that a client should be able to use an interface without needing to know the classes that implement that interface.
- The Open/Closed rule states that programs should be structured so that they can be revised by creating new classes instead of modifying existing ones.
- The Liskov Substitution Principle (LSP) specifies when it is meaningful for one interface to be a subtype of another. In particular, X should be a subtype of Y only if an object of type X can always be used wherever an object of type

Y is expected.

- The rule of Abstraction states that a class's dependencies should be as abstract as possible.