Java Program Design - Anotações Ch. 1

Data: 16/02/2023 & 17/02/2023
Tags: #SoftwareEngineering #java #POO
PDF: Java Program Design Principles, Polymorphism, and Patterns (Edward Sciore).pdf

---

> *How to use the object oriented programming facilities of Java effectively*

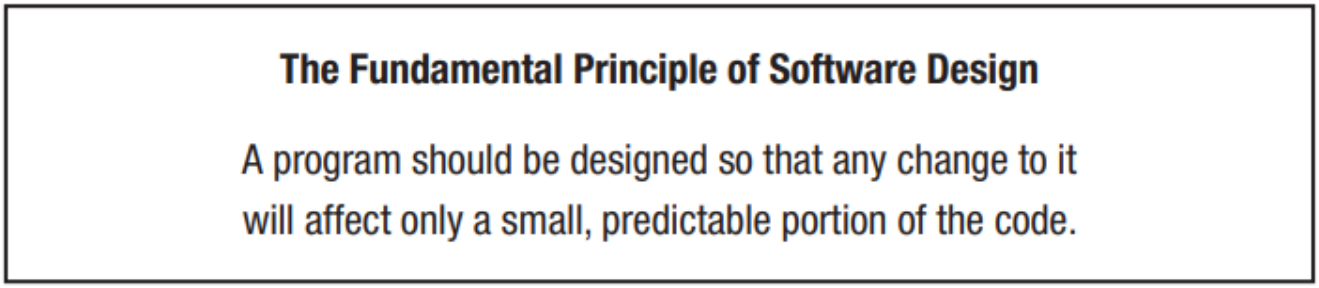
> This book can help you make sense of the Java library. Many of its classes are organized around design patterns, and understanding the pattern helps to understand why the classes are the way they are and how they should be used.

> Design patterns were introduced by Gamma, Held, Johnson, and Vlissides in their seminal 1994 book. The Java language has several features that support the use of these patterns. For example, the Iterable and Iterator interfaces support the iterator pattern, the Observable class and Observer interface support the observer pattern, and the Stream and Consumer interfaces support the visitor pattern. In addition, the availability of lambda expressions in Java makes it possible to reformulate several design patterns more simply and organically.

Modular Software Design

When a beginning programmer writes a program, there is one goal: the program must work correctly. However, correctness is only a part of what makes a program good. Another, equally important part is that the program be **maintainable**.

Suppose that you need to modify a line of code in a program. You will also need to modify the other lines of code that are impacted by this modification, then the lines that are impacted by those modifications, and so on. As this proliferation increases, the modification becomes more difficult, time-consuming, and error prone. **Therefore, your goal should be to design the program such that a change to any part of it will affect only a small portion of the overall code:**



If a variable is declared outside of a class then it can be referenced from any of the class's methods. It is said to have *global* scope. If the variable is declared within a method then it can be referenced only from within the code block where it is declared, and is said to have *local* scope.

```
public class ScopeDemo {
        private int x = 1;

        public void f() {
                int y = 2;
                for (int z = 3; z < 10; z++) {
                        System.out.println(x+y+z);
                }
                // ...
        }

        public void g() {
```

```
                int y = 7;
                // ...
        }
}
```

> Suppose that I decide to modify ScopeDemo so that the variable y in method f has a different name. Because of y's scope, I know that I only need to look at method f, even though a variable named y is also mentioned in method g. On the other hand, if I decide to rename variable x then I am forced to look at the entire class.

In general, the smaller the scope of a variable, the fewer the lines of code that can be affected by a change. Consequently, the fundamental design principle implies that each variable should have the smallest possible scope.

Object-Oriented Basics

Each object belongs to a *class*, which defines the object's capabilities in terms of its public variables and methods.

APIs and Dependencies

The public variables and methods of a class are called its Application Program Interface (or API).

The designer of a class is expected to document the meaning of each item in its API. Java has the Javadoc tool specifically for this purpose. The Java 9 class library has an extensive collection of Javadoc pages, available at the URL.

This class is *client* of another *class* (which has `.length()`, etc.)

```
public class StringCLient {
        public static void main(String[] args) {
                String s = "abc";
                System.out.println(s.length());
        }
}
```

> The code for StringClient implies that the class String must have a method length that satisfies its documented behavior. However, the StringClient code has no idea of or control over how String computes that length.

If X is a client of Y then Y is said to be a dependency of X. The idea is that X depends on Y to not change the behavior of its methods. If the API for class Y does change then the code for X may need to be changed as well.

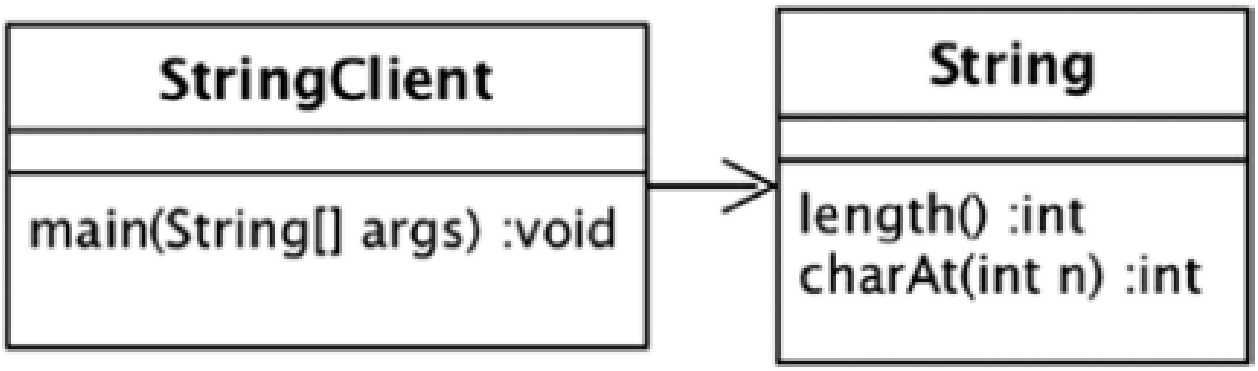
Modularity

Treating an API as a contract simplifies the way that large programs get written. A large program is organized into multiple classes. Each class is implemented independently of the other classes, under the assumption that each method it calls will eventually be implemented and do what it is expected to do. When all classes are written and debugged, they can be combined to create the final program.

This design strategy has several benefits. Each class will have a limited scope and thus will be easier to program and debug. Moreover, the classes can be written simultaneously by multiple people, resulting in the program getting completed more quickly. We say that such programs are modular. Modularity is a necessity; good programs are always modular. However, modularity is not enough. There are also important issues related to the design of each class and the connections between the classes. The design rules later in this chapter will address these issues.

Class Diagrams

Example:



Static vs. Nonstatic

A *static* variable is a variable that belongs to a class. It is shared among all objects of the class. If one object changes the value of a static variable then all objects see that change. On the other hand, a *nonstatic* variable "belongs" to an object of the class. **Each object has its own instance of the variable**, whose value is assigned independently of the other instances.

```
public class StaticTest {
        private static int x;
        private int y;

        public StaticTest(int val) {
                x = val;
                y = val;
        }

        public void print() {
                System.out.println(x + " " + y);
        }

        // not allowed to reference nonstatic variable
        public static int getX() {
                return x;
        }

        public static void main(String[] args) {
                StaticTest s1 = new StaticTest(1);
                s1.print(); // prints "1 1"

                StaticTest s2 = new StaticTest(2);
                s2.print(); // prints "2 2"
                s1.print(); // prints "2 1"
        }
}
```

Methods can also be static or nonstatic. A static method (such as getX in StaticTest) is not associated with an object. A client can call a static method by using the class name as a prefix. Alternatively, it can call a static method the conventional way, prefixed by a variable of that class.

> In Java, a static method is a method that **belongs to a class rather than to an instance of the class**. This means that you can call a static method on the class itself, without needing to create an object of that class.
>
> You don't need to create an instance of `MyClass` to call the static method.
> Static methods are often used for utility functions or helper methods that don't need to access any instance

> variables of the class. Since static methods belong to the class itself, they can be called even if no objects of the class have been created.
>
> Can be called without creating an instance.

For example, the two calls to getX in the following code are equivalent. To my mind, the first call to getX is to be preferred because it clearly indicates to the reader that the method is static.

```
StaticTest s1 = new StaticTest(1);
int y = StaticTest.getX();
int z = s1.getX();
```

A Banking Demo

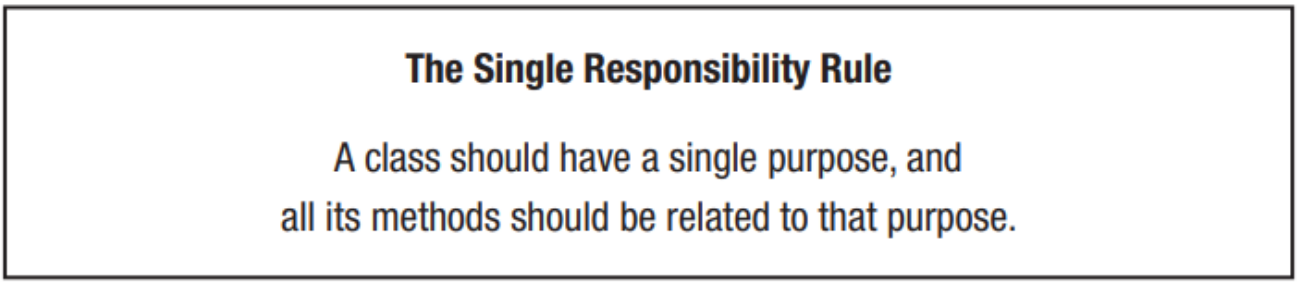
The code consists of a single class, named `BankProgram` - holds a map that stores the balances of several accounts held by a bank. Each element in the map is a key-value pair. The key is an integer that denotes the account number and its value is the balance of that account, in cents.

- The *quit* method sets the global variable done to true, which causes the loop to terminate.
- The global variable *current* keeps track of the current account.
- The *newAccount* method allocates a new account number, makes it current, and assigns it to the map with an initial balance of 0.
- The *select* method makes an existing account current. It also prints the account balance.
- The *deposit* method increases the balance of the current account by a specified number of cents.
- The method *authorizeLoan* determines whether the current account has enough money to be used as collateral for a loan. The criterion is that the account must contain at least half of the loan amount.
- The *showAll* method prints the balance of every account.
- Finally, the *addInterest* method increases the balance of each account by a fixed interest rate.

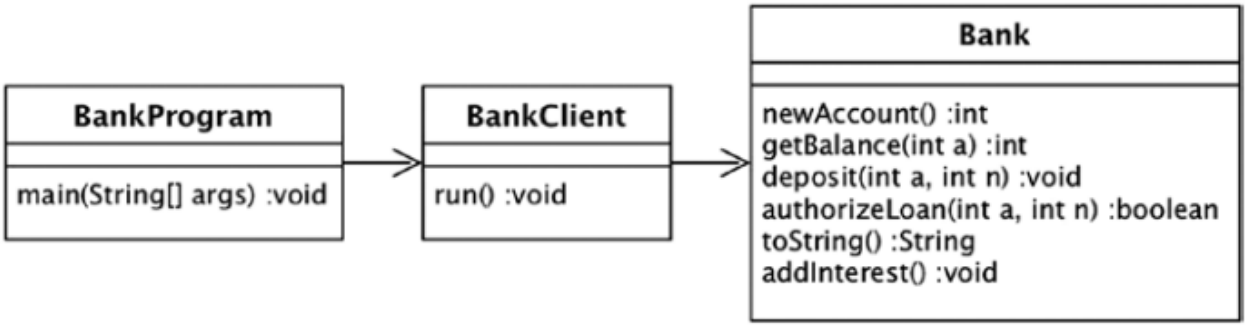
See version 1.

The Single Responsibility Rule

The `BankProgram` code is correct, But is it any good? Note that the program has multiple areas of responsibility - for example, one responsibility is to handle I/O processing and other responsability is to manage account information - and both responsibilities are handled by a single class.



Version 2 of the banking demo is an example of such a design. It contains three classes: The class `Bank` is responsible for the banking information; the class `BankClient` is responsible for I/O processing; and the class `BankProgram` is responsible for putting everything together.


BankProgram
main(String[] args) :void
BankClient
run() :void
Bank
newAccount() :int
getBalance(int a) :int
deposit(int a, int n) :void
authorizeLoan(int a, int n) :boolean
toString() :String
addInterest() :void

```
private void newAccount() {
   current = bank.newAccount();
   System.out.println("Your new account number is "
                      + current);
}

private void select() {
   System.out.print("Enter acct#: ");
   current = scanner.nextInt();
   int balance = bank.getBalance(current);
   System.out.println("The balance of account "
                      + current + " is " + balance);
}

private void deposit() {
   System.out.print("Enter deposit amt: ");
   int amt = scanner.nextInt();
   bank.deposit(current, amt);
}
```

```
private void authorizeLoan() {
   System.out.print("Enter loan amt: ");
   int loanamt = scanner.nextInt();
   if (bank.authorizeLoan(current, loanamt))
      System.out.println("Your loan is approved");
   else
      System.out.println("Your loan is denied");
}

private void showAll() {
   System.out.println(bank.toString());
}
```

Refactoring

In general, to refactor a program means to make syntatic changes to it without changing the way it works.

Unit Testing

Earlier in this chapter I stated that one of the advantages to a modular program is that each class can be implemented and tested separately. This begs the question: How can ou test a class separate from the rest of the program?

The answer is to write a ***driver*** program for each class. The driver program calls the various methods of the class, passing them sample input and checking that the return values are correct. The idea is that the driver should test all possible ways that the methods can be used. Each way is called a *use case*.

```
public class BankTest {
    private static Bank bank = new Bank();
    private static int acct = bank.newAccount();

    public static void main(String[] args) {
        verifyBalance("initial amount", 0);
        bank.deposit(acct, 10);
        verifyBalance("after deposit", 10);
        verifyLoan("authorize bad loan", 22, false);
        verifyLoan("authorize good loan", 20, true);
    }
        // ...
```

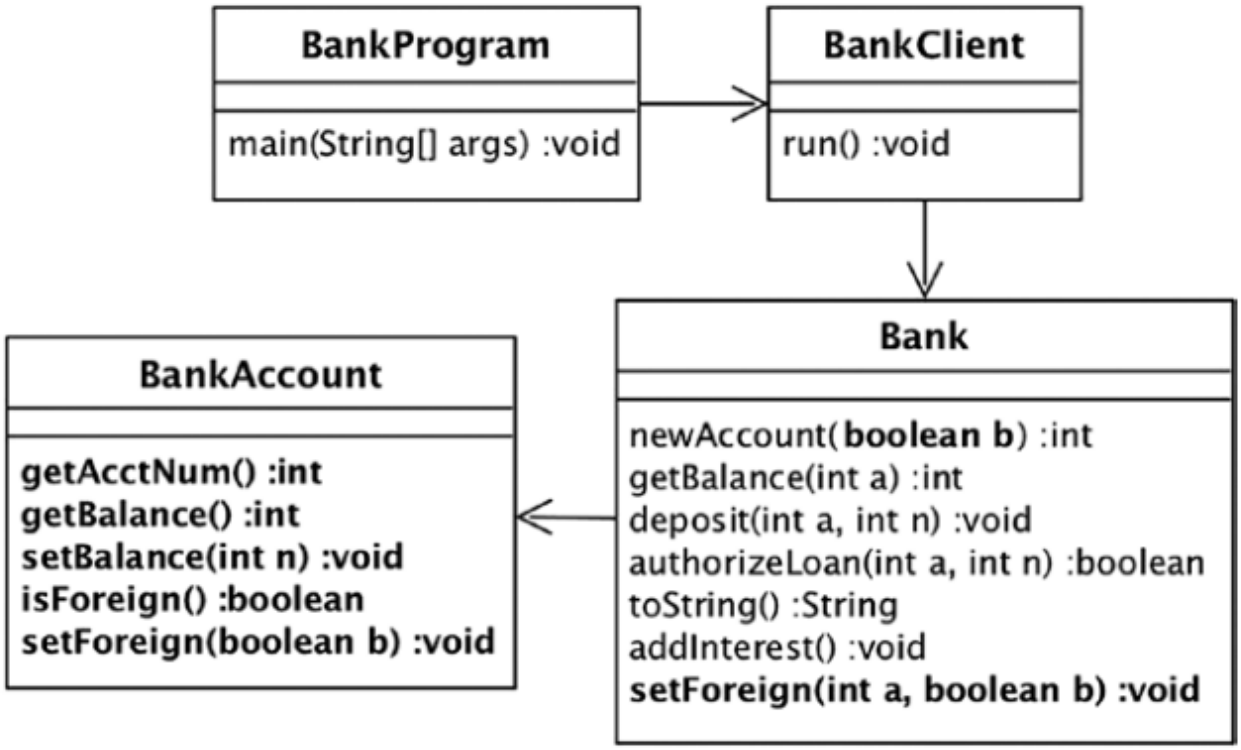
Class Design

A program that satisfies the Single Responsability rule will have a class ofr each identified responsability. But how do you know if you have identified all the responsibilities?

The short answer is that you don't. Sometimes, what seems to be a single responsibility can be broken down further. The need for a separate class may become apparent only when additional requirements are added to the program.

For example, consider version 2 of the banking demo. The class Bank stores its account information in a map, where the map's key holds the account numbers and its value holds the associated balances. **Suppose now that the bank also wants to store additional information for each account**. In particular, assume that the bank wants to know whether the owner of each account is foreign or domestic. How should the program change?

After some quiet reflection, you will realize that the program needs an explicit concept of a bank account. This concept can be implemented as a class; call it BankAccount. The bank's map can then associate a BankAccount object with each account number. These changes form version 3 of the banking demo.



> The class stores information of a bank account, and has setters (to change the information) and getters (to get the information)
> (Cada instância da class BankAccount tem acctnum, balance, e isforeign independentes visto que são apenas private e não private static)

```
public class BankAccount {
    private int acctnum;
    private int balance = 0;
    private boolean isforeign = false;

    public BankAccount(int a) {
        acctnum = a;
    }

    public int getAcctNum() {
        return acctnum;
    }

    public int getBalance() {
        return balance;
    }

    public void setBalance(int amt) {
        balance = amt;
    }

    public boolean isForeign() {
```

```
        return isforeign;
    }

    public void setForeign(boolean b) {
        isforeign = b;
    }
}
```

```
public class Bank {
   private HashMap<Integer,BankAccount> accounts
                                    = new HashMap<>();

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

 public void addInterest() {
    for (BankAccount ba : accounts.values()) {
       int balance = ba.getBalance();
       balance += (int) (balance * rate);
       ba.setBalance(balance);
    }
 }
```

Encapsulation

Let's look more closely at the code for BankAccount in Listing 1-10. Its methods consist of accessors and mutators (otherwise known as "getters" and "setters"). Why use methods? Why not just use public variables, as shown in Listing 1-13? With this class, a client could simply access BankAccount's variables directly instead of having to call its methods.

> An Alternative BankAccount Class

```
public class BankAccount {
        public int acctnum;
        public int balance = 0;
        public boolean isforeign = false;

        public BankAccount(int a) { acctnum = a; }
}
```

Although this alternative BankAccount class is far more compact, its design is far less desirable. Here are three reasons to prefer methods over public variables:

- methods are able to limit the power of clients. A public variable is equivalent to both an accessor and a mutator method, and having both methods can often be inappropriate. For example, clients of the alternative BankAccount class would have the power to change the account number, which is not a good idea.
- methods provide more flexibility than variables. Suppose that at some point after deploying the program, the bank detects the following problem: The interest it adds to the accounts each month is calculated to a fraction of a penny, but that fractional amount winds up getting deleted from the accounts because the balance is stored in an integer variable
- methods can perform additional actions. For example, perhaps the bank wants to log each change to an account's balance. If BankAccount is implemented using methods, then its setBalance method can be modified so that it writes to a log file. If the balance can be accessed via a public variable then no logging is possible.

> **The Rule of Encapsulation**
>
> A class's implementation details should be hidden from its clients as much as possible.

Redistributing Responsibility

The classes of the version 3 banking demo are modular and encapsulated.
Nevertheless,, there is something unsatisfactory about the design of their methods. In particular, the `BankAccount` methods don't do anything interesting. All the woek occurs in `Bank`.

For example, consider the action of depositing money in an account. The `bank`'s `deposit` method controls the processing. The `BanAccount` object manages the getting and setting of the bank balance, but it does so under the strict supervision of the `Bank` object.

This lack of balance between the two classes hints at a violation of the Single Responsibility rule. The intention of the version 3 banking demo was for the Bank class to manage the map of accounts and for the BankAccount class to manage each individual account. However, that didn't occur—**the Bank class is also performing activities related to bank accounts**.

Consider what it would mean for the BankAccount object to have responsibility for deposits. It would have its own deposit method:

```
public void deposit (int amt) {
        balance += amt;
}
```

And the `Bank`'s deposit method would be modified so that it called the `deposit` method of `BankAccount`:

```
public void deposit (int acctnum, int amt) {
        BankAccount ba = accounts.get(acctnum);
```

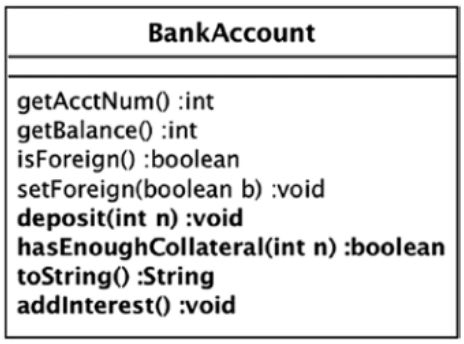
```
        ba.deposit(amt);
}
```

In this version, `Bank` no longer knows how to do deposits. Instead, it delegates the work to the appropriate `BankAccount` objetc.

Which version is a better design? The BankAccount object is a more natural place to handle deposits because it holds the account balance. Instead of having the Bank object tell the BankAccount object what to do, it is better to just let the BankAccount object do the work itself. We express this idea as the following design rule, called the Most Qualified Class rule.

> **The Most Qualified Class Rule**
>
> Work should be assigned to the class that knows best how to do it.

Version 4:




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

    public BankAccount (int acctnum) {
        this.acctnum = acctnum;
    }

    // ..

    public void deposit (int amt) {
        balance += amt;
    }

    public boolean hasEnoughCollateral (int loanamnt) {
        return balance >= loanamnt / 2;
    }

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

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

```
    }
}
```

> **Note**
>
> In Java, "this" is a keyword that refers to the current object instance of a class. It can be used within the methods or constructors of a class to refer to the fields or methods of that particular instance of the class.
>
> When an object is created from a class, it has its own unique set of field values. The "this" keyword is used to differentiate between the field of the current object and other fields with the same name in the class.
>
> For example, in the constructor of the BankAccount class, the "this" keyword is used to refer to the "acctnum" field of the current object instance and assign it the value of the parameter "acctnum". This is necessary because the parameter "acctnum" and the field "acctnum" have the same name, so without "this" keyword, the parameter would be assigned to itself instead of the field.
>
> In summary, the "this" keyword is used to refer to the current instance of an object in order to access or modify its fields and methods.

```
public class Bank {
    private HashMap<Integer,BankAccount> accounts;
    private int nextacct;

    public Bank (HashMap<Integer,BankAccount> accounts, int n) {
        this.accounts = accounts;
        nextacct = n;
    }

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

    public int getBalance (int acctnum) {
        BankAccount ba = accounts.get(acctnum);
        return ba.getBalance();
    }

        // ...
```

Dependency Injection

The Most Qualified Class rule can also be applied to the question of how to initialize the dependencies of a class. For example consider the `BankClient` class, which has dependencies on `Scanner` and `Bank`. The relevant code looks like this:

```
public class BankClient {
        private Scanner scanner = new Scanner(System.in);
        private Bank bank = new Bank();
        // ...
}
```

When the class creates its Scanner object it uses System.in as the source, indicating that input should come from the console. But why choose System.in? There are other options. The class could read its input from a file instead of the console or it could get its input from somewhere over the Internet. Given that the rest of the BankClient code does not

care what input its scanner is connected to, restricting its use to System.in is unnecessary and reduces the flexibility of the class.

A similar argument could be made for the bank variable. Suppose that the program gets modified so that it can access multiple banks. The BankClient code does not care which bank it accesses, so how does it decide which bank to use? The point is that BankClient is not especially qualified to make these decisions and therefore should not be responsible for them. Instead, some other, more qualified class should make the decisions and pass the resulting object references to BankClient. This technique is called dependency injection.

The point is that BankClient is not especially qualified to make these decisions and therefore should not be responsible for them. Instead, some other, more qualified class should make the decisions and pass the resulting object references to BankClient. This technique is called dependency injection. Typically, the class that creates an object is most qualified to initialize its dependencies. In such cases an object receives its dependency values via its constructor. This form of dependency injection is called constructor injection.

```
public class BankClient {
        private int current = -1;
        private Scanner scanner;
        private boolean done = false;
        private Bank bank;

        public BankClient(Scanner scanner, Bank bank) {
                this.scanner = scanner;
                this.bank = bank;
        }
        // ...
}
```

The class Bank can be improved similarly. It has one dependency, to its account map, and it also decides the initial value for its nextacct variable:

```
public class Bank {
        private HashMap accounts = new HashMap<>();
        private int nextacct = 0;
        // ...
}
```

The Bank object creates an empty account map, which is unrealistic. In a real program the account map would be constructed by reading a file or accessing a database. As with BankClient, the rest of the Bank code does not care where the account map comes from, and so Bank is not the most qualified class to make that decision. A better design is to use dependency injection to pass the map and the initial value of nextacct to Bank, via its constructor

```
public class Bank {
        private HashMap accounts;
        private int nextacct;

        public Bank(HashMap accounts, int n) {
        this.accounts = accounts;
        nextacct = n;
        }

        //...
}
```

The version 4 BankProgram class is responsible for creating the Bank and BankClient classes, **and thus is also responsible for initializing their dependencies**.

```
public class BankProgram {
        public static void main(String[] args) {
                HashMap accounts = new HashMap<>();
                Bank bank = new Bank(accounts, 0);
                Scanner scanner = new Scanner(System.in);
                BankClient client = new BankClient(scanner, bank);
                client.run();
        }
}
```

It is interesting to compare versions 3 and 4 of the demo in terms of when objects get created. In version 3 the BankClient object gets created first, followed by its Scanner and Bank objects. The Bank object then creates the account map. In version 4 the objects are created in the reverse order: first the map, then the bank, the scanner, and finally the client. This phenomenon is known as dependency inversion — **each object is created before the object that depends on it**.

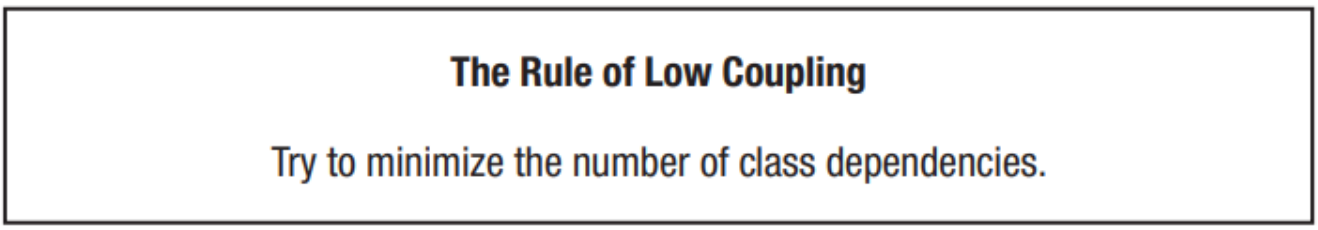
=> Note how BankProgram makes all the decisions about the initial state of the program. Such a class is known as a configuration class. A configuration class enables users to reconfigure the behavior of the program by simply modifying the code for that class.

> Note
>
> The idea of placing all dependency decisions within a single class is powerful and convenient. In fact, many large programs take this idea one step further. They place all configuration details (i.e., information about the input stream, the name of the stored data file, etc.) into a configuration file. The configuration class reads that file and uses it to create the appropriate objects.
>
> The advantage to using a configuration file is that the configuration code never needs to change. Only the configuration file changes. This feature is especially important when the program is being configured by end users who may not know how to program. They modify the configuration file and the program performs the appropriate configurations.

Mediation

The `BankClient` class in the version 4 banking demo does not know about `BankAccount` objects. It interacts with accounts solely through methods of `Bank` class. The `Bank` class is called a *mediator*.

Mediation can enhance the modularity of a program. If the Bank class is the only class that can access BankAccount objects then **BankAccount is essentially private to Bank**.



This rule is often expressed less formally as “Don’t talk to strangers.” The idea is that if a concept is strange to a client, or difficult to understand, it is better to mediate access to it.

Another advantage to mediation is that the mediator can keep track of activity on the mediated objects. In the banking demo, Bank must of course mediate the creation of BankAccount objects or its map of accounts will become inaccurate. The Bank class can also use mediation to track the activity of specific accounts. For example, the bank could track deposits into foreign accounts by changing its deposit method to something like this:

```
public void deposit (int acctnum, int amt) {
        BankAccount ba = accounts.get(acctnum);
```

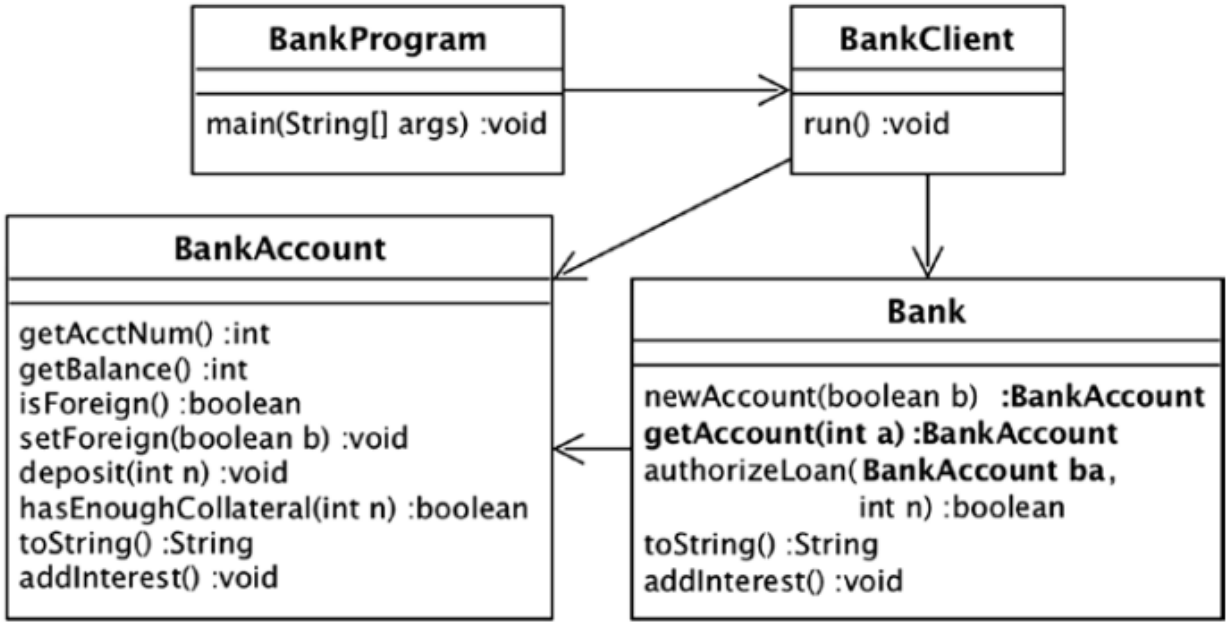
```
        if (ba.isForiegn())
                writeToLog(acctnum, amt, new Date());
        ba.deposit(amt);
}
```

Design Tradeoffs

The Low Coupling and Single Responsibility rules often **conflict** with each another.
Mediation is a common way to provide low coupling. But a mediator class tends to accumulate methods that are not central to its purpose, which can violate the Single Responsibility rule.

The banking demo provides an example of this conflict. The Bank class has methods getBalance, deposit, and setForeign, even though those methods are the responsibility of BankAccount. *But Bank needs to have those methods because it is mediating between BankClient and BankAccount.*

Another design possibility is to forget about mediation and let BankClient access BankAccount objects directly. A class diagram of the resulting architecture appears in Figure 1-5. In this design, the variable current in BankClient would be a BankAccount reference instead of an account number. The code for its getBalance, deposit, and setForeign commands can therefore call the corresponding methods of BankAccount directly. Consequently, Bank does not need these methods and has a simpler API. Moreover, the client can pass the reference of the desired bank account to the bank's authorizeLoan method instead of an account number, which improves efficiency.



Would this new design be an improvement over the version 4 banking demo? Neither design is obviously better than the other. Each involves different tradeoffs: Version 4 has lower coupling, whereas the new design has simpler APIs that satisfy the Single Responsibility rule better.

The Design of Java Maps

Each `HashMap` object is a mediator for its underlying `Map`