Java Program Design - Anotações Ch. 3

Data: 18/03/2023 && 06/04/2023
Tags: #SoftwareEngineering #java #POO
PDF: Java Program Design Principles, Polymorphism, and Patterns (Edward Sciore).pdf

---

Subclasses

Java allows one class to extend another. If class A extends class B, then A is said to be a *subclass* of B and B is a *superclass* of A. Subclass A inherits all public variables and methods of its superclass B, as well as all of B's code for these methods.

The most common example of subclassing in Java is the built-in class `Object`. By definition, every class in Java is a subclass of `Object`. That is, the following two class definitions are equivalent:

```
class Bank { ... }
class Bank extends Object { ... }
```

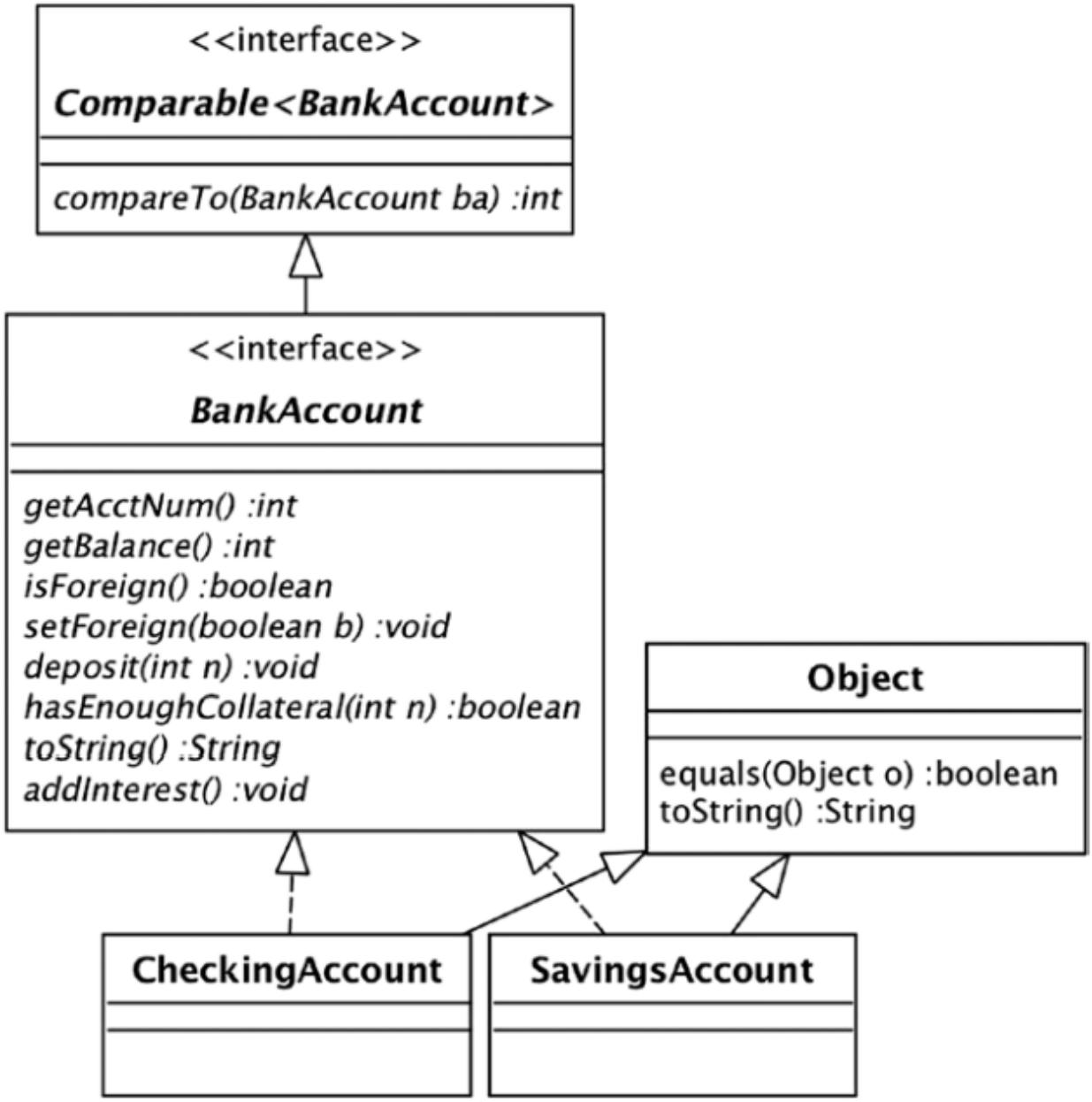


Figure 3-1. *Adding Object to a class diagram*

If A extends B then A **is a** B.

As an example, suppose that you want to modify the banking demo to have the new bank account type "interest checking." An interest checking account is exactly like a regular checking account except that it gives periodic interest. Call this class InterestChecking.

Should InterestChecking extend CheckingAccount? When I described interest checking I said that it "is exactly like" regular checking. This suggests an IS-A relationship, but let's be sure. Suppose that the bank wants a report listing all checking accounts. Should the report include the interest checking accounts? If the answer is "yes" then there is an IS-A relationship and InterestChecking should extend CheckingAccount. If the answer is "no" then it shouldn't.

```
public class InterestChecking extends CheckingAccount {
        private double rate = 0.01;

        public InterestChecking (int acctnum) {
                super(acctnum);
        }
```

```
        public String toString() {
                return "Interest checking account " + super.getAcctNum(); // ...
        }

        public void addInterest() {
                int newbalance = (int) (getBalance() * rate);
                deposit(newbalance);
        }
}
```

Note that the constructor calls the method **super**. The super method is a call to the superclass's constructor and is used primarily when the subclass needs the superclass to handle its constructor's arguments. If a subclass's constructor calls super then Java requires that the call must be the first statement of the constructor.

The private variables of a class are not visible to any other class, including its subclasses. This forces the subclass code to access its inherited state by calling the superclass's public methods. For example, consider again the proposed InterestChecking code of Listing 3-3. The toString method would like to access the variables acctnum, balance, and isforeign from its superclass. However, these variables are private, which forces toString to call the superclass methods getAcctNum, getBalance, and isForeign to get the same information. Similarly, the addInterest method has to call getBalance and deposit instead of simply updating the variable balance.

A protected variable is accessible to its descendent classes in the hierarchy but not to any other classes. For example, if CheckingAccount declares the variable balance to be protected then the addInterest method of InterestChecking can be written as follows:

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

Abstract Classes

Consider again version 6 of the banking demo. The CheckingAccount and SavingsAccount classes currently have several identical methods. If these methods need not remain identical in the future then the classes are designed properly. However, **suppose that the bank's policy is that deposits always behave the same regardless of the account type. Then the two deposit methods will always remain identical; in other words, they contain duplicate code.**

The existence of duplicate code in a program is problematic because this duplication will need to be maintained as the program changes. For example, if there is a big fix to the `deposit` method of `CheckingAccount` then you will need to remember to make the same bug fix to `SavingsAccount`.

The "Don't Repeat Yourself" Rule

A piece of code should exist in exactly one place.

The DRY rule is related to the Most Qualified Class rule, which implies that a piece of code should only exist in the class that is most qualified to perform it. If two classes seem equally qualified to perform the code then there is probably a flaw in the design – **most likely, the design is missing a class that can serve as the most qualified one**. In Java, a common way to provide this missing class is to use an **abstract class**.

Version 6 of the banking demo illustrates a common cause of duplicate code: two related classes implementing the same interface. A solution is to create a superclass of `CheckingAccount` and `SavingsAccount` and move the duplicate methods to it, together with the state variables they use. Call this superclass `AbstractBankAccount`. The classes `CheckingAccount` and `SavingsAccount` will each hold their own class-specific code and will inherit their remaining code from `AbstractBankAccount`. This design is version 7 of the banking demo. The code for `AbstractBankAccount` appears in Listing 3-4. This class contains

- the state variables `acctnum`, `balance`, and `isforeign`. These variables have the `protected` modifier so that the subclasses can access them freely.
- a constructor that initializes `acctnum`. This constructor is protected so that it can only be called by its subclasses (via their `super` method).
- code for the common methods `getAcctNum`, `getBalance`, `deposit`, `compareTo`, and `equals`.

```
package javaprogramdesign.chapter03.bank07;

public abstract class AbstractBankAccount implements BankAccount {
   protected int acctnum;
   protected int balance = 0;
   protected boolean isforeign = false;

   protected AbstractBankAccount(int acctnum) {
      this.acctnum = acctnum;
   }

   public int getAcctNum() {
      return acctnum;
   }

   public int getBalance() {
      return balance;
   }

   public boolean isForeign() {
      return isforeign;
   }

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

   public void deposit(int amt) {
```

```
      balance += amt;
   }

   public int compareTo(BankAccount ba) {
      int bal1 = getBalance();
      int bal2 = ba.getBalance();
      if (bal1 == bal2)
         return getAcctNum() - ba.getAcctNum();
      else
         return bal1 - bal2;
   }

   public boolean equals(Object obj) {
      if (! (obj instanceof BankAccount))
         return false;
      BankAccount ba = (BankAccount) obj;
      return getAcctNum() == ba.getAcctNum();
   }

   public abstract boolean hasEnoughCollateral(int loanamt);
   public abstract String toString();
   public abstract void addInterest();
}
```

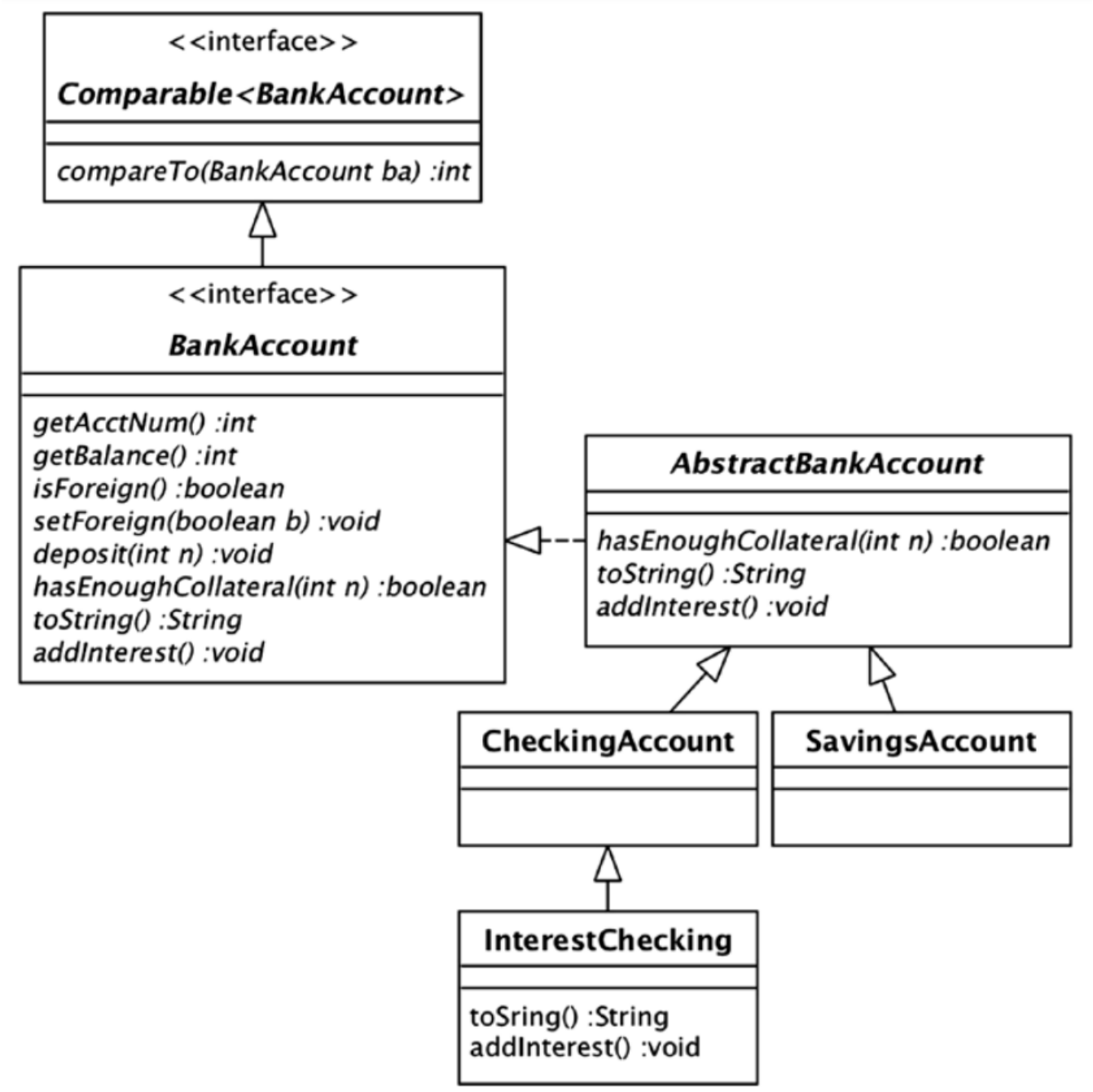
Note the declarations for the methods `hasEnoughCollateral`, `toString`, and `addInterest`. These methods are declared to be `abstract` and have no associated code. The issue is that `AbstractBankAccount` implements `BankAccount`, so those methods need to be in its API; however, the class has no useful implementation of the methods because the code is provided by its subclasses. By declaring those methods to be abstract, the class asserts that its subclasses will provide code for them.

A class that contains an abstract method is called an abstract class and must have the abstract keyword in its header. An abstract class cannot be instantiated directly. Instead, it is necessary to instantiate one of its subclasses so that its abstract methods will have some code. For example:

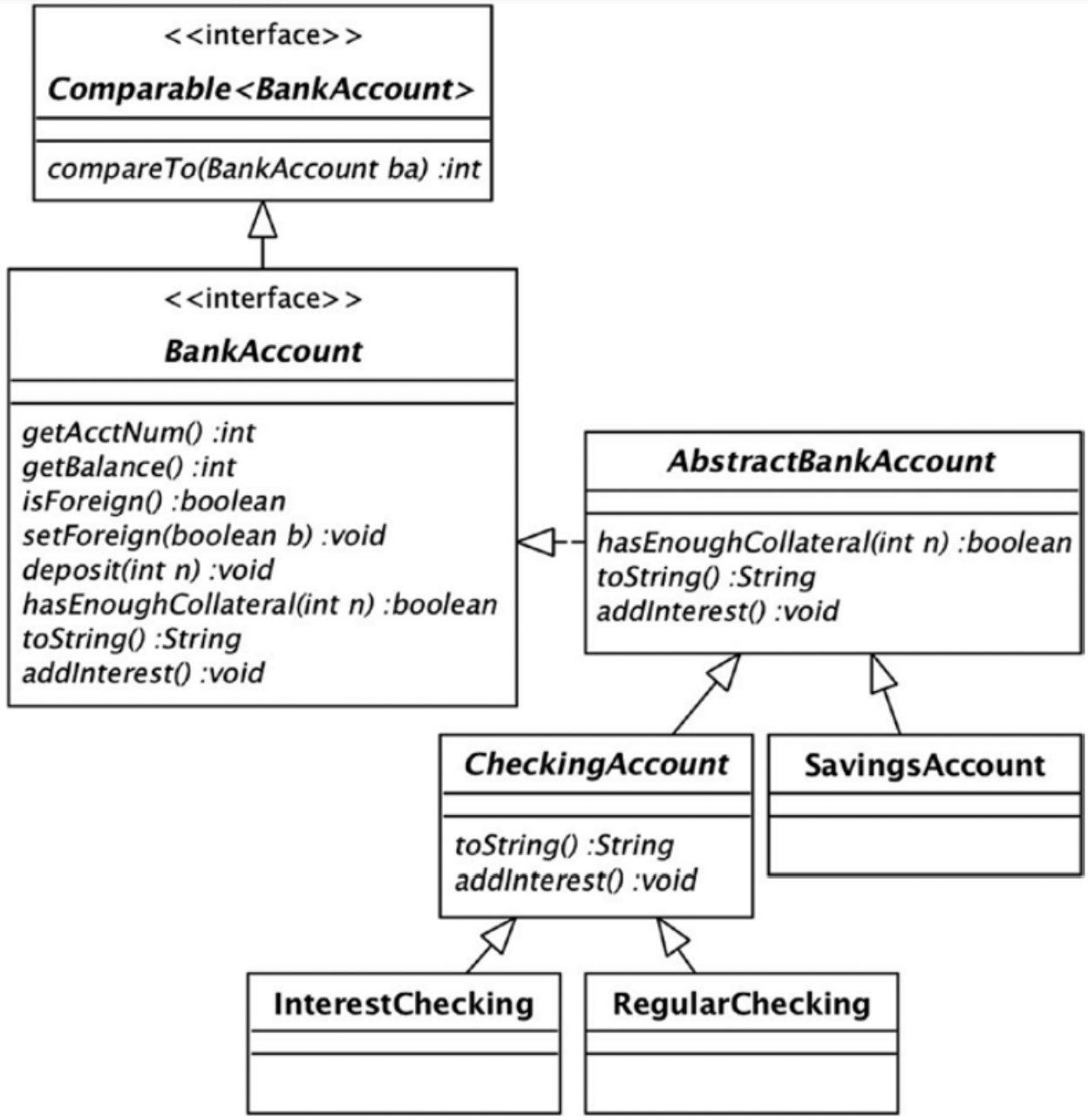
```
BankAccount xx = new AbstractBankAccount(123); // illegal
BankAccount ba = new SavingsAccount(123);      // legal
```

```
package javaprogramdesign.chapter03.bank07;

public class SavingsAccount extends AbstractBankAccount {
   private double rate = 0.01;

   public SavingsAccount(int acctnum) {
      super(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);
    }
}
```



On the other hand, a non-abstract superclass such as CheckingAccount plays both roles: it defines the category "checking accounts" (of which InterestChecking is a member) and it also denotes a particular member of that category (namely, the "regular checking accounts"). This dual usage of CheckingAccount makes the class less easy to understand and complicates the design. A way to resolve this issue is to split CheckingAccount into two pieces: an abstract class that defines the category of checking accounts and a subclass that denotes the regular checking accounts. Version 8 of the banking demo makes this change: the abstract class is CheckingAccount and the subclass is RegularChecking.




```
public abstract class CheckingAccount extends AbstractBankAccount {
        protected CheckingAccount (int acctnum) {
                super (acctnum);
        }

        public boolean hasEnoughCollateral(int loanamt) {
                return balance >= 2 * loanamt / 3;
        }

        public abstract String toString();
        public abstract void addInterest();
}
```

```
public class RegularChecking extends CheckingAccount {
        public RegularChecking (int acctnum) {
                super(acctnum);
        }

        public String toString() {
                return "Regular checking account " + acctnum;
```

```
        }

        public void addInterest() {
                // do nothing
        }
}
```

Writing Java Collection Classes

As an example, suppose that you want to create a class RangeList that implements List. A RangeList object will denote a collection that contains the n integers from `0` to `n-1`, for a value n specified in the constructor.

```
public class RangeListTest {

        public static void main(String[] args) {
                List<Integer> L = new RangeList(20);
                for (int x : L) {
                        System.out.print(x + " ");
                }
                System.out.println();
        }
}

public class RangeList extends AbstractList<Integer> {
        private int limit;

        public RangeList (int limit) {
                this.limit = limit;
        }

        public int size() {
                return limit;
        }

        public Integer get(int n) {
                return n;
        }
}
```

> output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Creating a scanner that reads from the file "testfile":

```
InputStream is = new FileInputStream("testfile");
Scanner sc = new Scanner(is);
```

The demo class EncryptDecrypt illustrates a typical use of byte streams. Its encrypt method takes three arguments: the name of the source and output files, and an encryption offset. It reads each byte from the source, adds the offset to it, and writes the modified byte value to the output. The main method calls encrypt twice. The first time, it encrypts the bytes of the file “data.txt” and writes them to the file “encrypted.txt”; the second time, it encrypts the bytes of “encrypted.txt” and writes them to “decrypted.txt.” Since the second encryption offset is the negative of the first, the bytes in “decrypted. txt” will be a byte-by-byte copy of “data.txt.

```
public class EncryptDecrypt {
        public static void main (String[] args) trows IOException {
```

```
                int offset = 26;
                encrypt("data.txt", "encrypted.txt", offset);
                encrypt("encrypted.txt", "decrypted.txt", -offset);
        }
        public static void encrypt (String source, String output,
                                                        int offset) throws IOException {
                try (InputStream is = new FileInputStream(source));
                         OutputStream os = new FileOutputStream(output);) {
                        int x;
                        while ((x = is.read()) >= 0) {
                                bytes b = (byte) x;
                                b += offset;
                                os.write(b);
                        }
                }
        }
}
```

The encrypt method illustrates the use of the read and write methods. The write method is straightforward; it writes a byte to the output stream. The read method is more intricate. It returns an integer whose value is either the next byte in the input stream (a value between 0 and 255) or a -1 if the stream has no more bytes. Client code typically calls read in a loop, stopping when the returned value is negative. When the returned value is not negative, the client should cast the integer value to a byte before using it.

```
public class BankProgram {
    public static void main(String[] args) {
        SavedBankInfo info = new SavedBankInfo("bank.info");
        Map<Integer,BankAccount> accounts = info.getAccounts();
        int nextacct = info.nextAcctNum();
        Bank bank = new Bank(accounts, nextacct);
        Scanner scanner = new Scanner(System.in);
        BankClient client = new BankClient(scanner, bank);

            client.run();
            info.saveMap(accounts, bank.nextAcctNum());
        }
    }
```

```
import java.io.*;
import java.nio.ByteBuffer;
import java.util.*;

public class SavedBankInfo {
   private String fname;
   private Map<Integer,BankAccount> accounts = new HashMap<Integer,BankAccount>();
   private int nextaccount = 0;
   private ByteBuffer bb = ByteBuffer.allocate(16);

   public SavedBankInfo(String fname) {
      this.fname = fname;
```

```
      if (!new File(fname).exists())
         return;
      try (InputStream is = new FileInputStream(fname)) {
         readMap(is);
      }
      catch (IOException ex) {
         throw new RuntimeException("bank file read exception");
      }
   }

   public Map<Integer,BankAccount> getAccounts() {
      return accounts;
   }

   public int nextAcctNum() {
      return nextaccount;
   }

   public void saveMap(Map<Integer,BankAccount> map, int nextaccount) {
      try (OutputStream os = new FileOutputStream(fname)) {
         writeMap(os, map, nextaccount);
      }
      catch (IOException ex) {
         throw new RuntimeException("bank file write exception");
      }
   }

   private void readMap(InputStream is) throws IOException {
      nextaccount = readInt(is);
      BankAccount ba = readAccount(is);
      while (ba != null) {
         accounts.put(ba.getAcctNum(), ba);
         ba = readAccount(is);
      }
   }

   private int readInt(InputStream is) throws IOException {
      is.read(bb.array(), 0, 4);
      return bb.getInt(0);
   }

   private BankAccount readAccount(InputStream is) throws IOException {
      int n = is.read(bb.array());
      if (n < 0)
         return null;
      int num       = bb.getInt(0);
      int type      = bb.getInt(4);
      int balance   = bb.getInt(8);
      int isforeign = bb.getInt(12);

      BankAccount ba;
      if (type == 1)
         ba = new SavingsAccount(num);
      if (type == 2)
         ba = new RegularChecking(num);
      else
         ba = new InterestChecking(num);
      ba.deposit(balance);
      ba.setForeign(isforeign == 1);
      return ba;
   }
```

```
   private void writeMap(OutputStream os, Map<Integer,BankAccount> map, int nextacct)
         throws IOException {
      writeInt(os, nextacct);
      for (BankAccount ba : map.values())
         writeAccount(os, ba);
   }

   private void writeInt(OutputStream os, int n) throws IOException {
      bb.putInt(0, n);
      os.write(bb.array(), 0, 4);
   }

   private void writeAccount(OutputStream os, BankAccount ba) throws IOException {
      int type = (ba instanceof SavingsAccount)  ? 1
            : (ba instanceof RegularChecking) ? 2 : 3;
      bb.putInt(0, ba.getAcctNum());
      bb.putInt(4, type);
      bb.putInt(8, ba.getBalance());
      bb.putInt(12, ba.isForeign() ? 1 : 2);
      os.write(bb.array());
   }
}
```

The Template Pattern

The abstract class will implement all the methods of its API, but not necessarily completely. The partially-implemented methods call "helper" methods, which are protected (that is, they are not visible from outside the class hierarchy) and abstract (that is, they are implemented by subclasses). This technique is called the template pattern. The idea is that each partial implementation of an API method provides a "template" of how that method should work. The helper methods enable each subclass to appropriately customize the API methods. In the literature, the abstract helper methods are sometimes called "hooks." The abstract class provides the hooks, and each subclass provides the methods that can be hung on the hooks. The version 8 BankAccount class hierarchy can be improved by using the template pattern. The problem with the version 8 code is that it still violates the DRY rule. Consider the code for method hasEnoughCollateral in the classes SavingsAccount and CheckingAccount. These two methods are almost identical. They both multiply the account balance by a factor and compare that value to the loan amount. Their only difference is that they multiply by different factors. How can we remove this duplication?

The solution is to **move the multiplication and comparison up to the AbstractBankAccount class and create an abstract helper method that returns the factor to multiply by.** This solution is implemented in the version 9 code. The code for the hasEnoughCollateral method in AbstractBankAccount changes to the following:

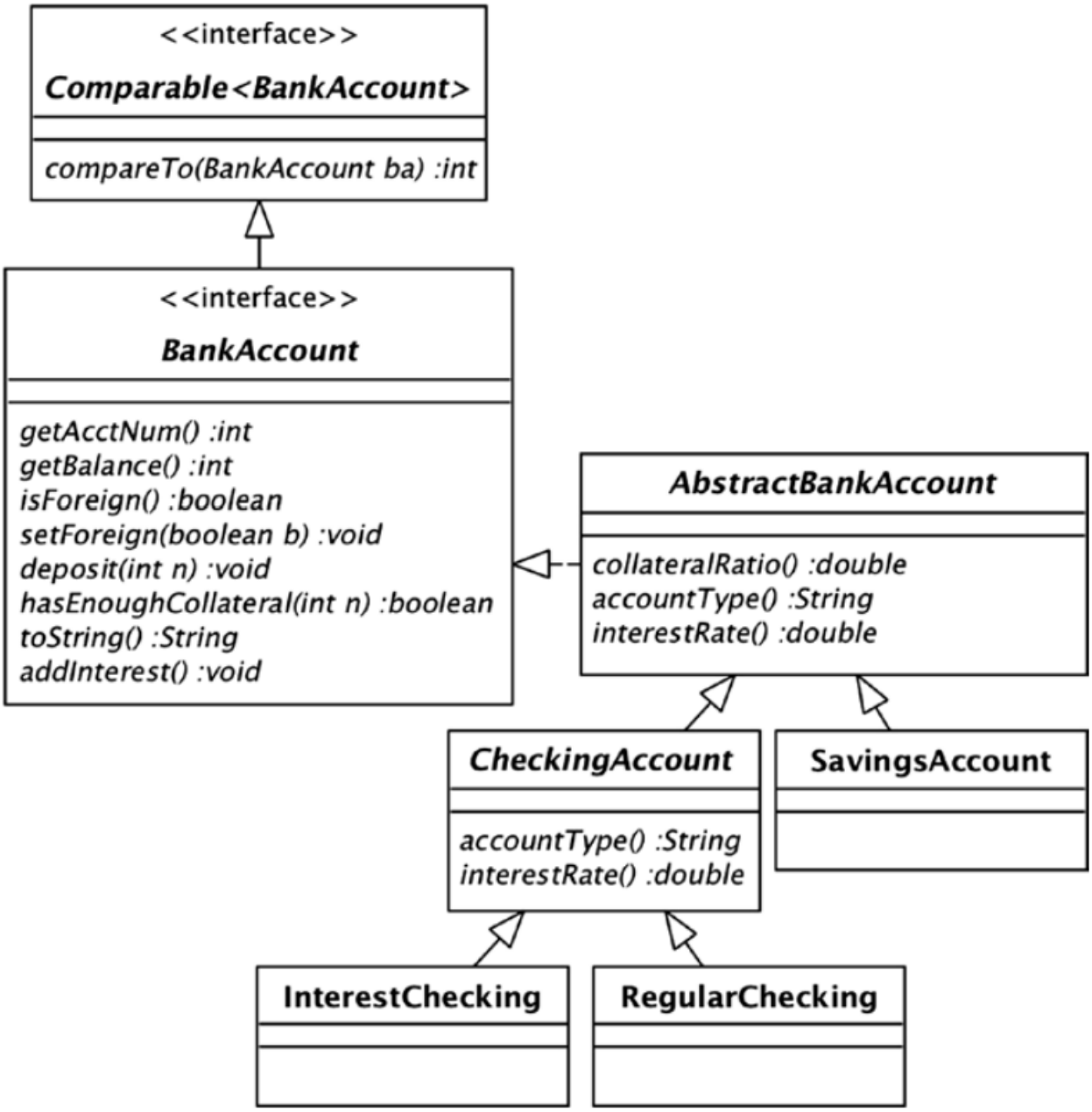
```
public boolean hasEnoughCollateral (int loanamt) {
        double ration = collateralRatio();
        return balance >= loanamt * ratio;
}

protected abstract double collateralRatio();
```

That is, the `hasEnoughCollateral` method is no longer abstract.
Instead, it is a template that calls the abstract helper method `collateralRatio`, whose code is implemented by the subclasses. For example, here is the version 9 code for the `collateralRatio` method in `SavingsAccount`.

```
protected double collateralRatio() {
        return 1.0 / 2.0;
}
```

The abstract methods addInterest and toString also contain duplicate code. Instead of having each subclass implement these methods in their entirety, it is better to create a template for them in AbstractBankAccount. Each

template method can call abstract helper methods, which the subclasses can then implement. In particular, the addInterest method calls the abstract method interestRate and the toString method calls the abstract method accountType.




```
public abstract class AbstractBankAccount implements BankAccount {
        protected int acctnum;
        protected int balance;

        // ...

        public String toString() {
                String accttype = accountType();
                return accttype +  " account"; // ...
        }

        // ...
        protected abstract double collateralRatio();
        protected abstract String accountType();
        protected abstract double interestRate();
}
```

```
public abstract class CheckingAccount extends AbstractBankAccount {

   protected CheckingAccount(int acctnum) {
      super(acctnum);
   }

   protected double collateralRatio() {
      return 2.0 / 3.0;
   }

   protected abstract String accountType();
   protected abstract double interestRate();
}
```

```
public class RegularChecking extends CheckingAccount {

   public RegularChecking(int acctnum) {
      super(acctnum);
   }

   protected String accountType() {
      return "Regular checking";
   }

   protected double interestRate() {
      return 0.0;
   }
}
```