Advanced Java Inheritance

The banking problem was originally written by Nick, and the code you see was written by Julie.

Banking Problem

Create an object-oriented design for the following problem concerning bank accounts. For each account, you must track the current balance as well as count the number of transactions per month and record the lowest daily balance. The goal for your design is to avoid duplicating code between the three types of accounts described below. All accounts needs to respond to the following messages:

```
Constructor
initialize a new account with an opening balance

void deposit(double amt)
add an amount to the balance and increment the number of
transactions

void withdraw(double amt)
subtract an amount to the balance and increment the number of
transactions

double getBalance();
return the current balance

void endMonth()
the account will be sent this message once a month, so it should
levy any monthly fees at that time and print out the account
monthly summary
```

There are three types of accounts:

MinBalance: Deposit and withdraw just affect the balance with no transaction fees. If the account balance ever drops below some required threshold, there is a $5.00 monthly fee levied at the end of the month.

Nickel 'n Dime: No monthly fee, but every withdrawal generates a $0.50 fee. The withdrawal of the fee shouldn't count as a separate transaction.

Gambler: A withdrawal returns the requested amount of money— however the amount deducted from the balance is as follows: there is a 0.49 probability that no money will actually be subtracted from the balance. There is a 0.51 probability that twice the amount actually withdrawn will be subtracted. Either way it counts as a transaction.

Propose some classes to store the idea of an account. You may use an abstract superclass or you may subclass some of the account from others.

Where are the instance variables declared?

Where do you use overriding?

How are you going to track the minimum monthly balance? What ramifications does this have for calling the inherited `withdraw` method versus just directly changing the `balance` instance variable? (Can you tell this example was originally an exam question? :-)

The `Account.java` file defines of the abstract superclass `Account`.

```
/*
 * Account class
 * -------------
 * The Account class is an abstract super class with the default
 * characteristics of a bank account. It maintains a balance and a
 * current number of transactions. It does not have an implementation
 * of endMonth() behavior which should be filled in by its subclasses.
 * Because it is declared as an abstract method, you cannot ever
 * instantiate an Account object, only subclasses which provide an
 * implementation of endMonth can be created.
 */

public abstract class Account  {

   protected Account(double openingBalance)           // constructor
   {
      balance = openingBalance;
      minBalance = balance;
      numTransactions = 0;
   }

   public double getBalance()
   {
      return balance;
   }

   public void withdraw(double amt)
   {
      balance -= amt;
      if (balance < minBalance)           // if we've hit a new low, record it
         minBalance = balance;
      numTransactions++;
   }

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

   // Factors some common behavior up which will occur in the subclasses endMonth
   protected void printMonthlyStatement()
   {
      System.out.println("Monthly statement");
      System.out.println("Num transactions this month: " + numTransactions);
      System.out.println("Lowest monthly balance: $" + minBalance);
      System.out.println("Current balance: $" + balance);
```

```
      minBalance = balance;         // reset to start new month
      numTransactions = 0;
   }

   abstract void endMonth();        // force subclasses to implement



   protected double balance;        // protected, so subclasses can access
   protected double minBalance;
   protected int numTransactions;

}
```

The subclasses:

```
public class MinBalance extends Account {

   public MinBalance(double openingBal)   // constructors are NOT inherited
   {
      super(openingBal);
   }

   public void endMonth()
   {
      if (minBalance < 500.00)
         balance -= 5.00;                // take out the monthly fee
      printMonthlyStatement();
   }

}

public class NickelNDime extends Account {

   public NickelNDime(double openingBal)  // constructors are NOT inherited
   {
      super(openingBal);
   }

   public void withdraw(double amt)
   {
      super.withdraw(amt);    // use superclass version of withdraw
      balance -= 0.50;        // subtract fee, but don't call withdraw
                              // to avoid trans count
   }

   public void endMonth()
   {
      printMonthlyStatement();
   }

}
```

```
public class Gambler extends Account {

   public Gambler(double openingBal)      // constructors are NOT inherited
   {
      super(openingBal);
   }

   public void withdraw(double amt)
   {
      if (Math.random() > .49)      // random number generator in Math class
         super.withdraw(2 * amt);   // unlucky!
      else
         super.withdraw(0.00);      // happy happy joy joy
   }

   public void endMonth()
   {
      printMonthlyStatement();
   }

}
```

A sample test `main` and its support methods:

```
   public class Main {

      public static void main(String args[])
      {
         Account[] accounts;
         int i;

         accounts = new Account[10];            // Creates array itself
         for (i = 0; i < accounts.length; i++)
            accounts[i] = RandomAccount();

         for (int day = 1; day < 31; day++) {
               int accountNum = randomInt(0, accounts.length-1);
            if (randomInt(0, 1) == 0)
               accounts[accountNum].deposit(randomInt(10,100));
            else
               accounts[accountNum].withdraw(randomInt(10,100));  // polymorphism!!
            }

         for (i = 0; i < accounts.length; i++) {
            System.out.print("\nAccount #" + i + " ");
            accounts[i].endMonth();
            }
      }
```

```
   /*
    * You could see how the switch statement below wouldn't scale well
    * especially if we added a lot of new account types.
    * Hang on and we'll see a better way to do this by exploiting
    * the fact that Java keeps class names around at RT...
    */

   private static Account RandomAccount()
   {
      switch (randomInt(1,3)) {
         case 1: return (new Gambler(500));  // initial starting balance of $500
         case 2: return (new NickelNDime(500));
         case 3: return (new MinBalance(500));
      }
      return null;   // never gets here, but compiler couldn't realize that
   }


   /*
    * This is a cover on Math.random to make it a little easier to use.
    * Where to put this? -- unclear.  There is a Random class but it
    * doesn't quite do what we need, so we just put our own little
    * wrapper into our Main class.  Sometimes OO doesn't always fit
    * what you need...
    */

   private static int randomInt(int low, int high)
   {
      int range = high - low + 1;
      return (int)(Math.random() * range) + low;
   }
}
```

Output:

```
Account #0 Monthly statement
Num transactions this month: 4
Lowest monthly balance: $500
Current balance: $556

Account #1 Monthly statement
Num transactions this month: 2
Lowest monthly balance: $407
Current balance: $502

Account #2 Monthly statement
Num transactions this month: 0
Lowest monthly balance: $500
Current balance: $500

Account #3 Monthly statement
Num transactions this month: 4
Lowest monthly balance: $429
Current balance: $461

Account #4 Monthly statement
Num transactions this month: 2
Lowest monthly balance: $500
Current balance: $549.5
```

```
   Account #5 Monthly statement
   Num transactions this month: 4
   Lowest monthly balance: $458
   Current balance: $464

   Account #6 Monthly statement
   Num transactions this month: 4
   Lowest monthly balance: $500
   Current balance: $528

   Account #7 Monthly statement
   Num transactions this month: 1
   Lowest monthly balance: $500
   Current balance: $510

   Account #8 Monthly statement
   Num transactions this month: 2
   Lowest monthly balance: $438
   Current balance: $438

   Account #9 Monthly statement
   Num transactions this month: 7
   Lowest monthly balance: $500
   Current balance: $673
```

Class variables and methods: the "static" keyword

Consider the `MinBalance` class. There is a threshold on the lowest allowable balance and a corresponding monthly fee levied when the balance falls under this. Throwing "magic numbers" into the code is really not an optimal solution for this. Instead, this is a perfect opportunity to introduce class-wide data using the `static` keyword (discussed in the Java objects handout) to store that data in a named location that can be retrieved and changed for all `MinBalance` objects in one place. Here is the `MinBalance` class with these added *class variables*:

```
public class MinBalance extends Account {

   public MinBalance(double openingBal)   // constructors are NOT inherited
   {
      super(openingBal);
   }

   public void endMonth()
   {
      if (minBalance < minBalanceRequired)
         balance -= monthlyFee;                // take out the monthly fee
      printMonthlyStatement();
   }

   protected static double minBalanceRequired = 500.00;
   protected static double monthlyFee = 5.00;
}
```

If we were to change the one `minBalanceRequired` variable, all `MinBalance` accounts would start using the new value, it requires no propagation.

How would we go about changing the fee or minimum requirement? It doesn't seem quite right to have an instance method for this, since it would require having an instance to message. Similar to declaring `static` variables, we can declare static methods to access this data. Add these *class methods* to the `MinBalance` class:

```
public static void setMinBalanceRequirement(double newBalance)
{
   minBalanceRequired = newBalance;
}

public static void setMonthlyFee(double fee)
{
   monthlyFee = fee;
}
```

Declaring these methods `static` indicates they will not be sent to a particular instance, they will be sent to the class itself using the name of the class as the receiver, like this:

```
MinBalance.setMinBalanceRequirement(1000);
```

From within a non-static instance method, you can access both `static` and non-`static` variables and `static` and non-`static` methods. However, within a `static` method, you cannot access any non-`static` variables or call any non-`static` methods. That makes sense because the receiver in a static method is not an object, the `this` is the class itself which only has access to the class-wide information.

Constant data and methods: the `final` keyword

And just what about making constants out of those magic numbers? The `final` keyword (discussed in the Java objects handout) is a way of marking a variable as "read-only". Its value is set once and then cannot be changed. For example, if we were committed to a constant transaction fee cost in the `NickleNDime` class, we could create a constant for it:

```
protected static final double transactionFee = 0.50;
```

The same access specifiers apply as usual. Note that Java doesn't allow you to add things to the upper-level namespace, so all data and methods must be defined within a class, including constant data. If you wish everyone to have access to a constant, you can mark it `public` so that everyone can read it. No one will be able to write it since it is `final`.

The `final` keyword can also be applied to methods, where it has a similar semantics—i.e. that the definition will not change. A `final` method cannot be overridden and changed by subclasses, this definition is the "`final`" one. This is used when you are concerned that a subclass may accidentally or deliberately wreak havoc by re-implementing an important method. A `final` method can also be handled slightly more efficiently since the code may be inlined and avoid the cost of the runtime dispatch.

You can also apply the `final` keyword to an entire class definition which means that the class cannot be subclassed. This is used for efficiency reasons on some of the built-in classes that don't have good potential as parent classes (such as the `Boolean`, `Integer`, etc. classes as well as `String`) Constants, defined using the `static final` keywords, are often used to create enumerated types since there is no `enum` facility in Java. You can adjust the visibility using the access specifiers to make the `enum` publicly accessible or just for the use of the class itself.

Class objects

As we've seen, an object always "knows" what type it is and ensures that all messages sent to it get mapped to the appropriate method during execution. By using some of the support supplied in the `Object` base class, from which all classes eventually derive, you can even ask an object about its class at runtime. The `instanceof` operator allows you to determine if an object is an instance of a particular class. An object is considered an instance of a class if it belongs to that class directly or belongs to one of its descendant classes (so, a `Boss` is considered an instance of `Employee`, for example).

```
Employee e;
Boss b;

if (e instanceof Boss) {
   System.out.println("It's a boss!");
   b = (Boss)e;
}
```

Sometimes `instanceof` is used before attempting a class cast to verify it is okay, since an incorrect cast will throw an exception at runtime.

We can also ask any object for its class using the `getClass` method which returns an object of type `Class`. (Classes are objects too!) A class can tell you its name and give you its superclass, and tell you what interfaces it implements:

```
Object obj;          // obj can be of any object type
Class cls;

cls = obj.getClass();
System.out.println("I'm a " + cls.getName());
```

Although these facilities allow you to explore the runtime type structure, you will rarely have a legitimate need to use this. For example, you **should not** use the runtime type structure to figure out what class an object is and then decide how to handle it, e.g. do not find out whether a `Shape` object is a `Square` or `Circle` so you can write a switch statement to call the right drawing routine. Use the runtime polymorphic dispatch for this: add a `draw` method to all the `Shape` classes and then when you message any `Shape` to `draw`, the object itself knows which method is the correct one for this class of `Shape`.

We can also look up classes by string name, using the `Class` `static` method `forName`:

```
Class cls;

cls = Class.forName("MinBalance");
```

And finally, we can ask a `Class` object to instantiate a new instance of that class using the `newInstance` method:

```
Class cls;
MinBalance acct;

cls = Class.forName("MinBalance");
acct = (MinBalance)cls.newInstance();
```

A few things to note: the assignment to `MinBalance` requires a cast since the `newInstance` method is defined to return something of `Object` type (since it has to work for all `Classes`, it can be no more specific than that) and using `newInstance` requires that the class has a default zero-arg constructor. This is the one used to set up the new object. There are a few other details about using the class methods that are a little unusual (it requires an exception handling domain to catch any problem), but to show you how we could create our random accounts by choosing a class name string from an array, look at this:

```
public static Account RandomAccount()
{
   String[] classNames = {"NickelNDime", "Gambler", "MinBalance"};
   int choice = randomInt(0,classNames.length-1);
   Account acct = null;

   try {
      acct = (Account) Class.forName(classNames[choice]).newInstance(); }
   catch (Exception e) {
      return null
   };

   acct.setInitialBalance(500.00);
   return acct;
}
```