# Lab 1: Inheritance

---

## Introduction

There are three basic capabilities that a modern object oriented programming language typically will have:

1. *Abstract Data Types* - The ability to create new data types (classes) which support information hiding (private data) and a public interface.
2. *Polymorphism* - The interpretation of a message is in the hands of its receiver. For example, the behavior of the `toString()` message depends on the class to which it is sent.
3. *Inheritance* - The mechanism that allows one class to share the methods and representation of another class.

One of the advantages of inheritance is code reuse. If we can define the behavior of a class A in terms of a previously defined class B, we can reduce the amount of code that we need to write for class B. In addition, by reusing the code from class B, we can reduce the amount of testing that we need to do for class A.

## Design of the `AccountKinds` Hierarchy

In the following section, we will discuss steps used in designing a hierarchy of classes. This discussion is not in depth and there are a number of books that focus solely on how to design object oriented programs including the proper use of class hierarchies and inheritance.

## The Problem Description

Suppose we have been given the task of creating a program that will keep track of all the accounts for a bank. There are a number of different kinds of accounts that the bank supports.

- Regular Account - This account charges a fee of which is the smaller of 10 or 10% of the balance at the end of the month. There is no interest. There is a penalty of 10.00 if the balance falls below a minimum of 500.00.

- Interest Account - This account charges a fee of which is the smaller of 10 or 10% of the balance at the end of the month. There is interest of 7% paid monthly. There is no minimum balance required.
- Checking Account - This account charges a fee of which is the smaller of 10 or 10% of the balance at the end of the month. There is annual interest of 7% paid monthly . There is a penalty of 10.00 if the balance falls below a minimum of 100.00. There is a charge of 0.10 for each transaction.
- CD Account - This account charges a fee of which is the smaller of 10 or 10% of the balance at the end of the month. There is interest of 15% paid yearly. There is no minimum balance required, but if there is a withdrawal before 12 months have gone by there will be a penalty of 20% of the current balance.

Each of these accounts has a personal identification number (PIN) with it to provide protection.

**Attributes and Methods**

Our first task is to identify the attributes and methods that each of the classes will need. At first look, we can identify the following attributes for each of the classes:

| Attributes for Regular Account | Type | Description |
|---|---|---|
| name | String | the name of the account holder |
| balance | double | balance in the account |
| pin | String | personal identification number |
| minimum balance | double | minimum balance for the account |
| penalty | double | penalty if balance falls below the minimum balance |

| Attributes for Interest Account | Type | Description |
|---|---|---|
| name | String | the name of the account holder |
| balance | double | balance in the account |
| pin | String | personal identification number |
| interest | double | yearly interest |

| Attributes for Checking Account | Type | Description |
|---|---|---|
| name | String | the name of the account holder |
| balance | double | balance in the account |
| pin | String | personal identification number |
| interest | double | yearly interest |
| minimum balance | double | minimum balance for the account |
| penalty | double | penalty if balance falls below the minimum balance |

| | | |
|---|---|---|
| transactions | int | the number of deposits and withdrawals in a month |

| Attributes for CD Account | Type | Description |
|---|---|---|
| name | String | the name of the account holder |
| balance | double | balance in the account |
| pin | String | personal identification number |
| interest | double | yearly interest |
| penalty | double | penalty if early withdrawal |
| months | int | number of months since the creation of the account |

All of these classes need to have basically the same methods

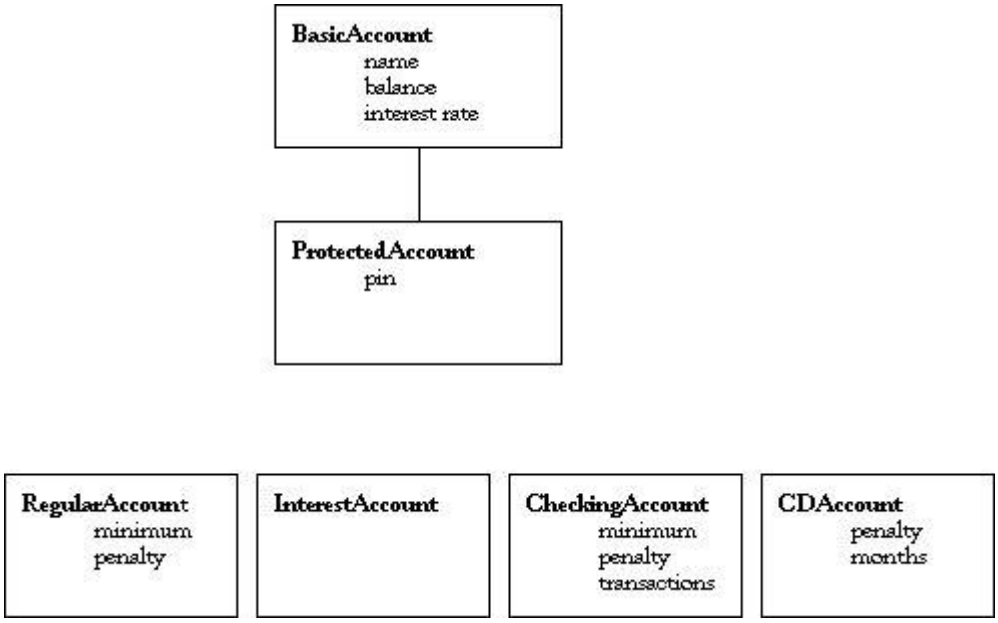
- Create the account.
- Deposit an amount.
- Withdraw an amount.
- Access the balance.
- Access the name.
- Check the validity of the PIN.
- Compute the fees.
- Compute the interest.

**Generalization and Specialization**

We could implement each of these classes separately. But that would mean that we would need to implement the same or similar code in each of the classes. We want to take the common code and attributes and place them in a single general class. This process is known as *generalization*. Examining the above tables, we see that all of the classes have name, balance, and pin. This suggests that we want a base class that encapsulates these attributes and the methods that work with them. We also notice that three of the four classes have interest. Since it is relatively easy to implement no interest as a rate of 0% and we expect most account classes to have interest, we will also add this into our base account.

Besides using generalization to decide on a class hierarchy, we often think in terms of layers of code. Thinking about our problem we realize that there are two kinds of things that our base class does. It provides features for handling the account (withdraw, deposit, compute interest) and it provides a security feature. In the future, we may wish to add additional security features like tracking the withdrawals to look for suspicious patterns. To make a place to implement those kinds of features, we will add a second class. It is in this class that we will put the pin number.

Currently our class hierarchy looks like:




`ProtectedAccount` will inherit the attributes of the `BasicAccount` and has an additional attribute.

The methods in `BasicAccount` will correspond to the list of methods we wrote down previously. The only methods that don't correspond are `monthly_update()` and `setRate()`. It is convenient to have the notion of a monthly update that we know every kind of account will respond to. In addition, we might want to eventually have an account for which the rate varies so we include a method by which we can change the rate. Our methods are:

```
BasicAccount ( String name)
String name ()
double balance ()
double rate ()
void deposit( double amount)
void withdraw( double amount)
void setRate( double rate)
void monthly_update()
double computeFees()
double computeInterest()
String toString()
```

We have to make some decisions about which of these methods should be accessible to the public. This is not a trivial decision. If we make a method public, then it will be inherited by every subclass and we are committed to those methods being in the

interface. Certainly it is appropriate for the accessor to name to be public. But what about `deposit()`? For this it is not so clear. In particular, we would like our `ProtectedAccount` to have two additional methods:

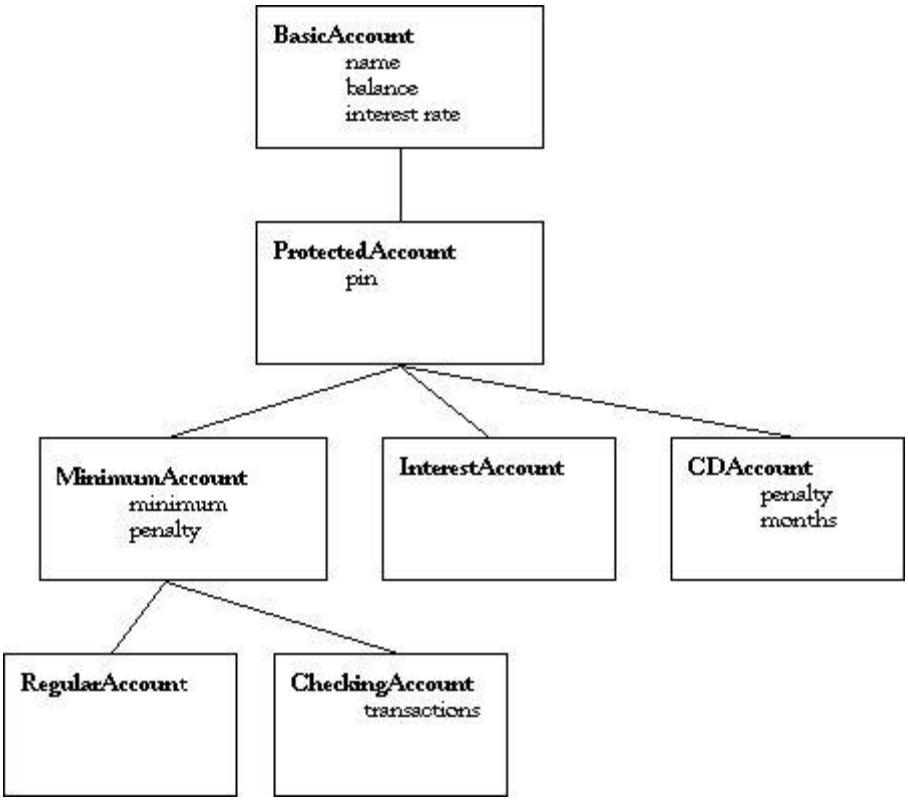
```
void deposit( double amount, String pin)
void withdraw( double amount, String pin)
```

which implement the checking of the pin number. If we make `deposit(double)` public, then everyone will be able to use this method and bypass the security. If we make `deposit(double)` private, then our subclass will not be able to use the method. We need another kind of protection. If we make the method `protected`, then it will be public for every subclass and private for all other classes. This is what we will do for the two methods `deposit()` and `withdraw()` in the `BasicAccount` class. We also do not want to allow others to be able to invoke the `setRate()` method and so will make that protected as well.

Question: "Do we want it to be possible for an instance of a `BasicAccount` to be created?"

Answer: No. We always want the protection services to be in place and therefore do not want instances of `BasicAccount` to be created. We can accomplish this by making the class `BasicAccount` to be `abstract`. If any code attempts to construct a `BasicAccount`, it will result in a compile time error.

If we look at our partial hierarchy, we see that there are a couple of classes that share the attributes minimum balance and penalty. Clearly these two classes can share these attributes and the code for computing the fees if the minimum is reached. We will generalize these into a new class `MinimumAccount` resulting in a new hierarchy:




At the top of the hierarchy is the most general class that we have. As we go farther down, the classes become more specialized. All of our classes commit themselves to the public interface in `BasicAccount` and `ProtectedAccount`. Do realize, however, that some of our subclasses will want to change the action of those methods. For example, if an instance of the `CDAccount` receives the `withdraw()` message, it will want to check and see if there is a penalty because of an early withdraw.

One of the advantages of have a class hierarchy like this is that we can write code that uses variables whose type is the generalized class. For example, we could create an array of type `BasicAccount`. Each of members of this array would have to be an instance of this class or *any* of its subclasses. We could safely loop over such an array and send each account the `monthly_update()` message. Each subclass is committed to implementing this method in an appropriate way for that class. Each instance will invoke the version of the method appropriate to the class of which it belongs.

## The Experimental Laboratory

Begin by creating a `Bank` project as appropriate for your environment. Copy the file `AccountDemo.java` and the folder `AccountKinds` ( which contains the class files for `BasicAccount.java`, `ProtectedAccount.java`, `MinimumAccount.java`, and `RegularAccount.java` as well as classes for exceptions that our classes may throw)

into your project and add them as appropriate for your environment.
The `RegularAccount` class and all of its parent classes have been implemented and we will be using them as a *software laboratory* for the experiments in this exercise.

Before starting the experiments, look at the code in the four classes and try to get a general feel for how it operates. We will use the experiments below to explore it in more detail.

---

**Experiments**

Below is a list of the experiments available for this exercise. Your instructor will tell you which ones you can omit, if any.

For each experiment that you are to perform, click its link and print a hard copy of the resulting web page. Then record your experimental results on that hard copy.

Experiment 1: Abstract Classes

Experiment 2: Constructors for Subclasses

Experiment 3: Subclasses - Inheriting Methods

Experiment 4: Overriding Inherited Methods

Experiment 5: Using super

---

**Creating the CheckingAccount**

We would like to complete the code for the `CheckingAccount` class. There are a number of closely related files that comprise our hierarchy of bank accounts. To make it easier to deal with them as a whole, Java has the notion of a *package*. All of the classes in a package will reside in a folder with the same name as the package. In our case, the package name is `AccountKinds`. Create a new file called `CheckingAccount.java` and place it in the `AccountKinds` folder.

The first thing to do is to start with a minimal class definition. Type in the following and personalize it.

```
/* CheckingAccount.java is an account that has a minimum balance
 * and 7% interest and a fee for each transaction of 10 cents.
 *
 * Written by: Charles Hoot, for Hands On Java.
 * Date: July 2000
 * Completed by:
 * Date:
 ****************************************************************/

package AccountKinds;

public class CheckingAccount extends MinimumAccount
{

}
```

The package statement identifies the package that this code belongs to. The extends tells us that `MinimumAccount` is the superclass of `CheckingAccount`.

If you compile this code, you should get an error message like the following:

```
Error : No constructor matching MinimumAccount() found in class
AccountKinds.MinimumAccount.
```

**Adding in the Attributes**

The problem is that Java knows that it needs to invoke a constructor for `MinimumAccount` when it makes `CheckingAccount`. If you do not create a constructor, Java will attempt to construct the super class using a constructor with no arguments. Since there is no such constructor in the `MinimumAccount` class, we get the above error.

Before we can complete the constructor, we need to determine the attributes required for this class. According to our class hierarchy, the only new attributes that this class requires are the number of transactions in the month.

As a matter of design we will also add in some static (shared by all members of the class) constant values for the minimum balance (`MINIMUM`), penalty amount ( `PENALTY`), interest rate ( `INTEREST_RATE`), and transaction cost (`TRANSACTION_COST`). This makes those values easy to change if needed.

Declare and initialize the four constants and create a private attribute named `myTransactions`.

Compile the code and continue when there are no new error messages.

**Creating the Constructor**

First lets add a stub for our constructor. Add the following code to the class.

```
    /*************************************************************
     * CheckingAccount constructor
     *
     * Input: name of account holder, and PIN, and initial balance
     *************************************************************/
    public CheckingAccount ( String name, String pin, double initial)
{

    }
```

Our constructor has some basic tasks it must accomplish. The very first thing that it must do is to invoke the constructor of the super class using `super()`. Look at the definition of the constructor in the `MinimumAccount` class and fill in the arguments for:

```
super(  ,   ,   ,   );
```

The remaining tasks that it needs to do are:

- Make the initial deposit.
- Set the interest rate.
- Initialize its private variable.

Complete code for these tasks and compile your code. You should not get any syntax errors.

Add in the following lines of code to `AccountDemo.java` and check to see that your code works as expected.

```
theScreen.print("\nCreating a Checking Account: ");
CheckingAccount anotherAccount;
anotherAccount = new CheckingAccount("The lone ranger", "9999", 200.0);

theScreen.println("\nAccount status is: " + anotherAccount);

theScreen.print("\nDoing a monthly update: ");
anotherAccount.monthly_update();
theScreen.println("\nAccount status is: " + anotherAccount);
```

We expect that the balance after a month will be 191.16.

### Overriding `withdraw()` and `deposit()`

We need to change what the `withdraw()` method does. It needs to do the regular `withdraw()` operations and the additional operation of increasing the number of transactions by one. To do this we need a way of invoking the `withdraw()` of our

superclass. This can be done via the use of `super` as is demonstrated in the following line:

```
super.withdraw(amount,pin);
```

Create methods for `deposit()` and `withdraw()` which first call the superclass method and then increment `myTransactions` by one.

Add in the following lines of code to `AccountDemo.java` and check to see that your code works as expected.

```
theScreen.println("Depositing 10.00 ");
anotherAccount.deposit(10.00, "9999");
theScreen.println("\nAccount status is: " + anotherAccount);

theScreen.println("Depositing 20.00 ");
anotherAccount.deposit(20.00, "9999");
theScreen.println("\nAccount status is: " + anotherAccount);

theScreen.println("Withdrawing 10.00 ");
anotherAccount.withdraw(10.0, "9999");
theScreen.println("\nAccount status is: " + anotherAccount);

theScreen.print("\nDoing a second monthly update: ");
anotherAccount.monthly_update();
theScreen.println("\nAccount status is: " + anotherAccount);
```

We expect that the balance after a second month the balance will be 202.398. This does not yet take into account the cost for the transactions yet.

**Overriding computeFees()**

We need to change what the `computeFees()` method does. As before, we need to use the super class method for `computeFees()` and to that result we need to add the transaction cost. Don't forget to reset the number of transactions for the next month.

Create a method for `computeFees()`. Then test the code. The balance should be .30 less than the previous result.

**Phrases you should now understand:**

Inheritance, Polymorphism, Generalization, Class Hierarchy, Subclass, Superclass, Package, Overriding a Method, Protected Attributes and Methods