DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 281 NOTES
Spring 2019: Amber Settle

Wednesday, May 8, 2019

Announcements

- Discuss the fourth assignment
- The fifth assignment is due tonight at 9 pm – questions?
- We have a make-up class this Friday
- The sixth and seventh assignments will be due at the usual times/days

Inheritance

Derived classes are defined in terms of a **base class** using **extends**. (Other terminology includes super/subclass or a parent/child class).

The derived class inherits the data members and methods of the base class. It can add other members to this, but this is what is provided initially.

This sort of situation is called **single inheritance**. Each class can have only one base class. (Multiple inheritance is not supported in Java).

Bank accounts

To illustrate inheritance, consider creating a class to represent a bank account.

Example: **Account.java** and **AccountDriver.java**

This class represents the actions and items that are common to all bank accounts. These include:

1. A name on the account, an account number, and a balance
2. A method to deposit money to the account
3. A method to withdraw money from the account
4. A method to see the current status of the account

Suppose that we want to create a specialized type of account, a **savings account** that will earn a specified amount of interest.

We can use the existing Account class to save some work, since many of the things that are found in the Account class are also things that a savings account needs to have (e.g. name, deposit, withdraw, display).
We will create a derived class of the Account class.

See **SavingsAccount.java**:

- The SavingsAccount class adds an interestRate instance variable and an addMonthlyInterest method
- Demonstrate how it works using the driver program
- The methods display and deposit are inherited from the Account class.
- The Account class does not have the addMonthlyInterest method.

We can also create a specialized account to represent a **checking account** that charges a monthly fee if you do not maintain a minimum monthly balance.

Again it will be a derived class of Account since it also needs the basic elements provided by that class.

See **CheckingAccount.java**:

- The CheckingAccount class adds minBal and monthlyFee instance variables and a checkMonthlyFee method
- Demonstrate the class using the driver program

Notes about the three account classes:

1. For now they have duplicated code in the constructors. There is a way to combine them that we will see in time.
2. Everything is public right now. We will discuss visibility modifiers and inheritance as our next topic. We have made everything public to make things simple as we introduce these classes.
3. The methods not defined in derived classes are automatically executed from the base class. See the println statements for illustration.
4. The derived classes do not share the methods that each defines separately.
5. Both accounts have a monthly update, but the operations performed are different. We will learn later how to combine them in some way in the base class.

Visibility modifiers and inheritance

We created our account classes with entirely public instance variables and methods.

This meant that everything in the base class was visible to the derived class. We could access balance inside of both SavingsAccount and CheckingAccount.

If we had not made it private, the results would have been very different.

The **rules for visibility modifiers** in derived/base classes:

- Public data members and methods of the base class are visible and public in the derived class

- Private data members and methods of the base class are NOT visible in the derived class.

This means that a derived class is working like an outside program.

Note: Inherited methods are still able to access non-inherited private members, even though the derived class cannot look at them directly. (E.g. deposit can still see balance and CheckingAccount can use deposit).

How do we make **a member of a base class visible to and inherited by the derived class**, but still make it private to an outside program?

There is a third visibility modifier: **protected**.

A protected data member is only visible to classes in the same package as the derived class.

So we use protected inheritance for data members that should be accessible in the derived class but not to any outside program that imports the classes that you have written and placed in the package.

The catch: If you use the default package, where all classes in the same directory are considered to be in the same package. Using that structure, the protected modifier will work exactly the same way as public has so far.

Note: Protected members can be inherited but default ones cannot.

Packages

In order to simplify the situation, and to make use of the protected modifier, we need to consciously start using packages.

Example: Look at the accounts code and how it uses packages.

As long as the base class and its derived classes are in the same package (and the program using them is outside), the **protected modifier** will now work as advertised.

Example: Change Account to use protected modifiers. Demonstrate that the driver program no longer has access to balance, but the derived classes do.

Constructors

We needed the empty default constructor in the Account class.

Without it, the SavingsAccount or CheckingAccount classes will not compile. Why?

When we create an object from a derived class, the derived class constructor is called.

However, if you do not explicitly call a base class constructor within it, the base class default constructor is called first automatically. This guarantees the initialization of inherited instance variables.

If within the derived class constructor, you want to call some other, parameterized constructor, you must do it explicitly using super.

You must do this in the very first line of the derived class constructor. Putting it anywhere else will give an error.

Example: We will **change the constructors for our SavingsAccount and CheckingAccount classes** to use the parameterized constructor for the Account class.

Objects and variables

We can declare variables from both built-in types (called **variables**) and from classes (called **objects**).

Examples: int count = 0; String myStr = new String("Hello!");

So far we have been thinking about them in the same way, but there are some fundamental differences between them that are important once we start working with methods.

Representation in memory

A variable of a built-in type has a value assigned to a **fixed location**.

Example: Draw a picture for count.

An object variable (called a reference variable in the book), represents a **reference to some location(s) in memory** that were explicitly allocated to store that object. This is the meaning of the 'new' in the initial assignment.

Example: Draw a picture for myStr.

When we declare an object variable but do not initialize it with 'new' to refer to a location, its **stored value is a special value called 'null'**. It represents no object at all.

Example: String anotherStr; Draw the picture.

Object variables and assignments

When we assign one variable of a built-in type to another, the value stored in one's location is overwritten with the value stored in the other's location.

Example: int num1 = 12, num2 = -3;
num1 = num2;

Draw the picture!

Technically, the same thing happens when we assign one object variable to another, but the value that is stored there represents an address in memory where the object is stored, so we do not make a copy of the object, but rather set the second object variable to point to the same object as the first.

Example: Weight value1 = new Weight();
Weight value2 = new Weight(25.5, 'p');
value1 = value2;

Draw the picture!

Object comparisons

The explanation above tells us why we need to write methods for classes to test equality, rather than relying on the '= =' operator.

If we use the '= =' operator, it will compare the two references (not the two objects that they refer to), and will return true if they are pointing to the same object.

It is not good enough for it to be two different objects with the same data member values.

If we want to define a notion of equality for different objects from a class, we must write a method (for example called equals) that returns a boolean value to implement this kind of comparison.

Exercise: Write an **equals** method for the account classes. Should it belong in one of the subclasses (CheckingAccount or SavingsAccount) or in the Account class? Why?

Printing objects

The System.out.println method can handle variables from any built-in data type without problems. This is not the case for user-defined objects.

Objects can be complicated, so System.out.println does not know exactly what you want to print when you send it an object.

It handles it by simply printing the name of the object.

Example: Modify **AccountDriver.java** to include a print statement.

To get around this, we will add a method to our class with the following header:

```
public String toString()
```

where the body takes whatever parts of the object you want to be displayed in the output and assembles them into a single string which is returned by the method.

When an object from the class is encountered in a call to System.out.println (or System.out.print), **toString will automatically be called** on that object to create an appropriate String for output.

Example: Write a **toString** method for the account classes. Where should the toString method go: in the Account class or in the child classes (CheckingAccount and SavingsAccount) or both? Why?

If we put it in all of the classes, can we make the subclasses use the toString method of the Account class?