برنامه نویسی پیشرفته
C#

۲۸ آبان ۹۸

ملکی مجد

Case Study

- Friendly Bank

- You are taking the role of a programmer who will be using the language to create a solution for a customer.
- We will be creating a bank application using C# and will be exploring the features of C#

Bank System Scope

- we are simply concerned with managing the account information in the bank.
- This information includes
    - their name, address, account number, balance and overdraft value
    - …
- There are also a number of different types of accounts
- The system must also generate warning letters and statements as required

Enumerated Types

```
enum AccountState
{
    New,
    Active,
    UnderAudit,
    Frozen,
    Closed
}
```

```
struct Account {
     public AccountState State;
     public string Name ;
     public string Address ;
     public int AccountNumber ;
     public int Balance ;
     public int Overdraft ;
}

Account RobsAccount;
```

- *Code Sample 23 Generous Account Structure*

```
const int MAX_CUST = 100;
Account [] Bank = new Account [MAX_CUST];
```

```
Bank[0] = RobsAccount;
Bank [25].Name;
```

Putting account information into arrays

```
class AccountStructureArray {
   public static void Main() {
      const int MAX_CUST = 100;
      Account[] Bank = new Account[MAX_CUST];
      Bank[0].Name = "Rob";
      Bank[0].State = AccountState.Active;
      Bank[0].Balance = 1000000;
      Bank[1].Name = "Jim";
      Bank[1].State = AccountState.Frozen;
      Bank[1].Balance = 0;
   }
}
```

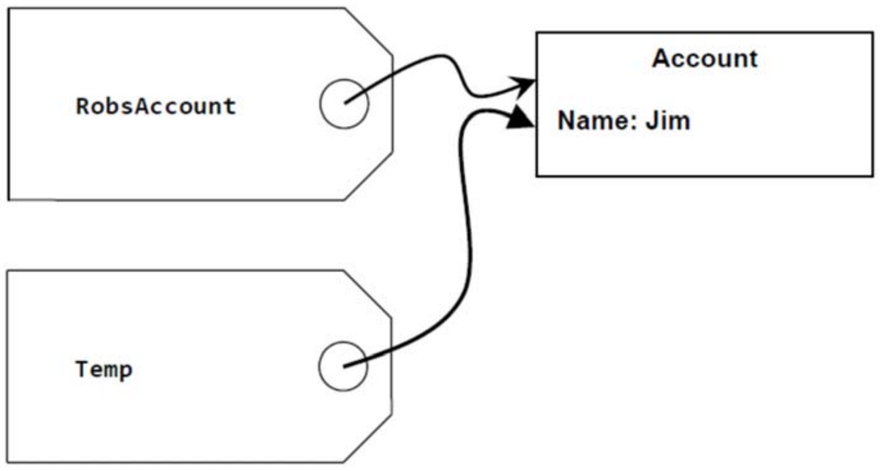
Non-compiling Account class

```
class Account {
    public string Name ;
}
class StructsAndObjectsDemo {
    public static void Main () {
        Account RobsAccount ;
        RobsAccount.Name = "Rob";
        Console.WriteLine (RobsAccount.Name );
    }
}
```

Compiling Account Class

```
class Account {
      public string Name ;
} ;
class StructsAndObjectsDemo {
      public static void Main () {
            Account RobsAccount ;
            RobsAccount = new Account();
            RobsAccount.Name = "Rob";
            Console.WriteLine (RobsAccount.Name );
      }
}
```

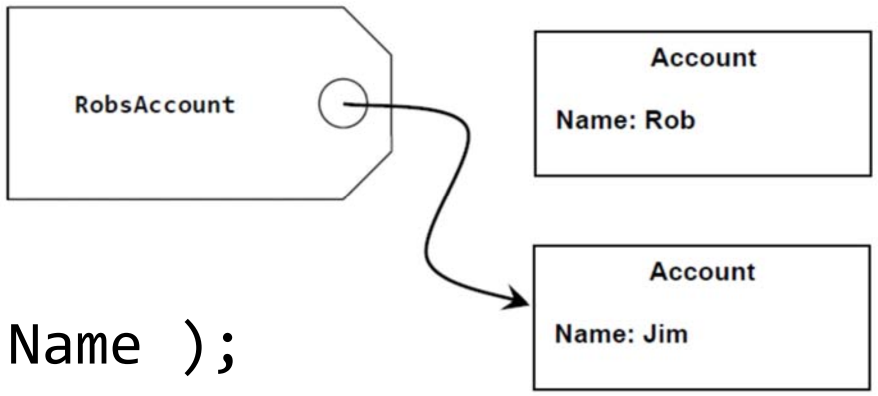
Multiple References



- `Account RobsAccount ;`
- `RobsAccount = new Account();`
- `RobsAccount.Name = "Rob";`
- `Console.WriteLine (RobsAccount.Name );`
- `Account Temp ;`
- `Temp = RobsAccount;`
- `Temp.Name = "Jim";`
- `Console.WriteLine (RobsAccount.Name );`

No References to an Instance

```
Account RobsAccount ;
RobsAccount = new Account();
RobsAccount.Name = "Rob";
Console.WriteLine (RobsAccount.Name );
RobsAccount = new Account();
RobsAccount.Name = "Jim";
Console.WriteLine (RobsAccount.Name );
```



there are a number of things that we need to be able to do with the bank account :

- pay money into the account
- draw money out of the account
- find the balance
- print out a statement
- change the address of the account holder
- print out the address of the account holder
- change the state of the account
- find the state of the account
- change the overdraft limit
- find the overdraft limit

Data in Objects

```
class Account {
    public decimal Balance;
}
Account RobsAccount ;
RobsAccount = new Account();
RobsAccount.Balance = 99;
```

RobsAccount.Balance = 0;

Member Protection inside objects

```
class Account {
private decimal balance;
}
```

Code Sample 31 Withdraw insufficient funds

- *Code Sample 32 Testing the Account Class*

Test Driven Development

- You don't do the testing at the end of the project
- You can write code early in the project which will probably be useful later on
- When you fix bugs in your program you need to be able to convince yourself that the fixes have not broken some other part

جلسه فردا را شرکت کنید

Using a static data member of a class

```
public class Account {
   public decimal Balance ;
   public static decimal InterestRateCharged ;
}
```

Account RobsAccount = new Account();

RobsAccount.Balance = 100;

Account.InterestRateCharged = 10;

Using a static method in a class

- we might have a method which decides whether or not someone is allowed to have a bank account.

- Make it static:
  - the method is part of the class, not an instance of the class.

```
public static bool AccountAllowed ( decimal income, int age )
{
if ( ( income >= 10000 ) && ( age >= 18 ) )
      { return true; }
else
      { return false; }
}
…………………………………………..
if ( Account.AccountAllowed ( 25000, 21 ) )
{
      Console.WriteLine ( "Allowed Account" );
}
```

Constructor

- The Default Constructor
- Our Own Constructor
- Feeding the Constructor Information
- Overloading Constructors
- Constructor Management

```
public Account (string inName, string inAddress,
  decimal inBalance)
{
  name = inName;
  address = inAddress;
  balance = inBalance;
}

public Account (string inName, string inAddress)
{
  name = inName;
  address = inAddress;
  balance = 0;
}

public Account (string inName)
{
  name = inName;
  address = "Not Supplied";
  balance = 0;
}
```

```
public Account (string inName, string inAddress,
  decimal inBalance)
{
  name = inName;
  address = inAddress;
  balance = inBalance;
}
public Account ( string inName, string inAddress ) :
  this (inName, inAddress, 0 )
{
}

public Account ( string inName ) :
  this (inName, "Not Supplied", 0 )
{
}
```

- *Code Sample 35 Overloaded Constructors*

A constructor cannot fail

```
public Account (string inName, string inAddress) {
if ( SetName ( inName ) == false )
{
      throw new Exception ( "Bad name " + inName) ;
}
if ( SetAddress ( inAddress) == false )
{
      throw new Exception ( "Bad address" + inAddress) ;
}
}
```

- *Code Sample 36 Constructor Failing*

Components and Hardware

- in a typical home computer, some parts are not "hard wired" to the system
  - the graphics adapter is usually **a separate device** which is plugged into the main board.
  - can buy a **new graphics** adapter at any time and fit it into the machine to improve the performance

  - For this to work properly the people who make main boards and the people who make graphics adapters have had to **agree on an *interface*** between two devices
  - **standard *interfaces*** which describe exactly how they fit together

Why we Need Software Components?

- A system designed without components is exactly like a computer with a graphics adapter which is part of the main board
    - not possible for me to improve the graphics adapter because it is "hard wired" into the system.

- For example, we might be asked to create a "**BabyAccount**" class which only lets the account holder draw out up to ten pounds each time. This might happen **even after we have installed** the system and it is being used.

Components and Interfaces

- An *interface* specifies how a software component could be used by another software component.

```
public interface IAccount {
   void PayInFunds ( decimal amount );
   bool WithdrawFunds ( decimal amount );
   decimal GetBalance ();
}
```

```
public class CustomerAccount : IAccount {
   private decimal balance = 0;
   public bool WithdrawFunds ( decimal amount ) {
      if ( balance < amount )
            { return false ; }
      balance = balance - amount ;
      return true;
   }
   public void PayInFunds ( decimal amount ) {
      balance = balance + amount ;
   }
   public decimal GetBalance () {
      return balance;
   }
   }
```

References to Interfaces

- CustomerAccount class
  - as a **CustomerAccount** (because that is what it is)
  - as an **IAccount** (because that is what it can do)

- Marzieh Malekimajd the individual (because that is who I am)
- A university lecturer (because that is what I can do)

- *Code Sample 38 Using Components*

Inheritance

- **Interface**: "I can do these things because I have told you I can"
  Inheritance: "I can do these things because my parent can"

```
public class BabyAccount : CustomerAccount,IAccount
{
}
```

Overriding methods

- The keyword **override** means "use this version of the method in preference to the one in the parent".
- The keyword virtual means “I might want to make another version of this method in a child class”. You don’t have to override the method, but if you don’t have the word present, you definitely can’t.
- This makes override and virtual a kind of matched pair. You use virtual to mark a method as able to be overridden and override to actually provide a replacement for the method.
- *Code Sample 39 Using Inheritance*