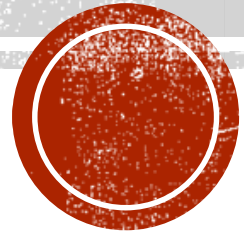
THE BANKINC APPLICATION



```
15 references
class SSIAAccount: Deposit
{

    0 references
    public SSIAAccount() { }
    6 references
    public SSIAAccount(double balance, string id, double intr):base(balance, id,intr)
    {

    }

    0 references
    public SSIAAccount(string id, double intr): base(id,intr)
    {

    }
    5 references
    public override void Withdraw(double amount)
    {
        Console.Write("not allowed on this account type");

    }

    3 references
    public override void Lodge(double amount)
    {
        balance += 1.25 * amount;
    }

    8 references
    public override string ShowDetails()
    {
        return String.Format("SSIAAccount id:{0}   Balance:{1,10}  Interest Rate{2,10}\n", id, balance.ToString("F2"), IntRate.ToString("F2"));

    }
}
```

BANKING APPLICATION

- To your Banking application add the following menu options:

- Option 1: Create account
- Option 2: Show All Account Details.
- Option 3: Lodge.
- Option 4: Withdraw.
- Option 5: Maintance: Run quarter interest rate
- Option 6: Exit

DATA STRUCTURE

- First we need a data structure to hold the bank accounts

```
List<Account> customersAccounts = new List<Account>();
```

LIST<T> CLASS

https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx

- <T> The type of elements in the list.
- Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.
- A dynamic data structure – unlike an array

Properties

Count	Gets the number of elements contained in the List<T>.

Methods

Add(T)	Adds an object to the end of the List<T>.
Clear()	Removes all elements from the List<T>.
Contains(T)	Determines whether an element is in the List<T>.
Remove(T)	Removes the first occurrence of a specific object from the List<T>.

THE MENU

```
Console.WriteLine("** the bank system **");


int option = 0;
do
{
    Console.WriteLine("1.   Create account");
    Console.WriteLine("2.  Show All Account Details");
    Console.WriteLine("3.  Lodge.");
    Console.WriteLine("4.  Withdrawal.");
    Console.WriteLine("5.  Maintance: Run quarter interest rate");
    Console.WriteLine("6.  Exit");
    option = Convert.ToInt32(Console.ReadLine());

    Console.WriteLine(option);
    switch (option)
    {
        case 1:
            {
                createAccounts(customersAccounts);
                break;
            }
        case 2:
            {
                displayAccounts(customersAccounts);
                break;
            }
```

MENU

```
                ;
            case 4:
                {
                    WithDraw(customersAccounts);
                    break;
                }
            case 6:
                break;

            default:
                {
                    Console.WriteLine("option not implemented ");
                    break;
                }
        }
    } while (option != 6);
}
```

OPTION 1

```
public static void createAccounts(List<Account> cAccounts)
{
    Deposit d1 = new Deposit(1000, "1001", 5);
    Deposit d2 = new Deposit(500, "1002", 3);
    Deposit d3 = new Deposit(200, "1003", 4);

    cAccounts.Add(d1);
    cAccounts.Add(d2);
    cAccounts.Add(d3);

    SSIAAccount SS1 = new SSIAAccount(100, "1004", 10);
    SSIAAccount SS2 = new SSIAAccount(1000, "1005", 10);
    SSIAAccount SS3 = new SSIAAccount(4000, "1008", 5);

    cAccounts.Add(SS1);
    cAccounts.Add(SS2);
    cAccounts.Add(SS3);

    CurrentAccount ca1 = new CurrentAccount(100,"1009", 300);
    CurrentAccount ca2 = new CurrentAccount(50,"1010", 100);
    CurrentAccount ca3 = new CurrentAccount(800,"1011", 400);

    cAccounts.Add(ca1);
    cAccounts.Add(ca2);
    cAccounts.Add(ca3);
    Console.WriteLine("accounts created on the system");


}
```

OPTION 2 – DISPLAY ALL BANK ACCOUNTS

```
1 reference
public static void displayAccounts(List<Account> cAccounts)
{
    foreach(Account a in cAccounts ){
        a.ShowDetails();
    }

}
```

OPTION 3 - LODGEMENT

```
public static void Lodge(List<Account> cAccounts)
{
    Console.WriteLine("process logdement - Enter in account id");
    String id = Console.ReadLine();
    Console.WriteLine("process logdement - Enter amount");
    double  amount = Convert.ToDouble(Console.ReadLine());

    Account fa = findAccount(cAccounts,id);
    if (fa != null)
    {
        fa.Lodge(amount);
    }
    else
    {
        Console.WriteLine("Error can't kind account");
    }

}
```

FIND ACCOUNT

```
    public static Account findAccount(List<Account> cAccounts, string id)
    {
        Account x = null;
        foreach (Account a in cAccounts)
        {
            if (a.id.Equals(id))
            {
                return a;
            }
        }

        return x;
    }

}
```

OPTION 4: WITHDRAW

```
public static void WithDraw(List<Account> cAccounts)
{
    Console.WriteLine("process withdrawal - Enter in account id");
    String id = Console.ReadLine();
    Console.WriteLine("process withdrawal - Enter amount");
    double amount = Convert.ToDouble(Console.ReadLine());

    Account fa = findAccount(cAccounts, id);
    if (fa != null)
    {
        fa.Withdraw(amount);
    }
    else
    {
        Console.WriteLine("Error can't kind account");
    }

}
```

POLYMORPHIC BEHAVIOUR

- Option 2, 3 and 4 all show polymorphic behaviour
- i.e. regardless of the account type(all different) the code is the same …
- Option 5 …. Is different …

OPTION 5 - DYNAMIC CASTING

```
public static void MaintanceRun(List<Account> cAccounts)
{

    foreach (Account a in cAccounts)
    {
        bool isDeposit = (a is Deposit);
        if (isDeposit)
        {

            Deposit d = (Deposit)a;
            d.AddQtrInterest();
            d.ShowDetails();
        }
    }
```

THE IS OPERATOR

Testing for type compatibility

The `is` keyword evaluates type compatibility at runtime. It determines whether an object instance or the result of an expression can be converted to a specified type. It has the syntax

C# Copy

```
expr is type
```

where *expr* is an expression that evaluates to an instance of some type, and *type* is the name of the type to which the result of *expr* is to be converted. The `is` statement is `true` if *expr* is non-null and the object that results from evaluating the expression can be converted to *type*; otherwise, it returns `false`.

For example, the following code determines if `obj` can be cast to an instance of the `Person` type:

C# Copy

```
if (obj is Person) {
   // Do something if obj is a Person.
}
```

AS OPERATOR

The `as` operator is like a cast operation. However, if the conversion isn't possible, `as` returns `null` instead of raising an exception. Consider the following example: