

# Industrial Programming

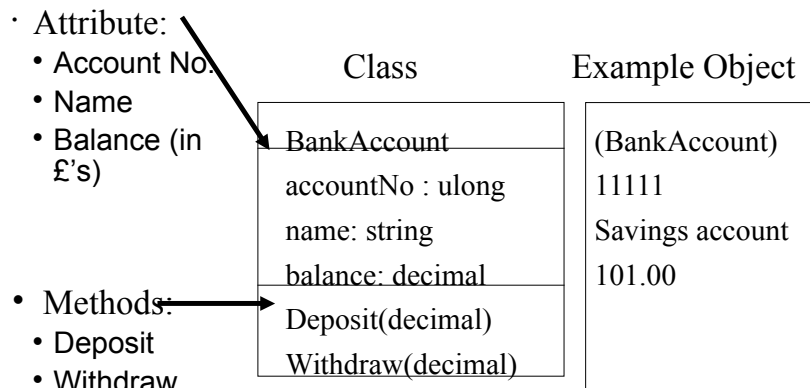
## Systems Programming & Scripting

### Lecture 12: C# Revision

# 3 Pillars of Object-oriented Programming

- **Encapsulation:** each class should be self-contained to localise changes. Realised through public and private attributes.
- **Specialisation:** model relationships between classes. Realised through inheritance.
- **Polymorphism:** treat a collection of items as a group. Realised through methods at the right level in the class hierarchy.

## Bank Account



## Extending the Example

- Define another class ProperBankAccount with an overdraft facility.
- Automatically assign account numbers to new accounts when generating them.
- Keep all field information secure within the account.

## Designing the Class

- Fields:
  - Those in `BankAccount` + overdraft
- Methods:
  - Those in `BankAccount` but modified
- Invariants:
  - Balance in `ProperBankAccount` is never lower than the negative overdraft.

## Revisiting the BankAccount Class

```
class BankAccount {  
    protected static ulong latestAccountNo = 1000;  
    protected ulong accountNo;  
    protected decimal balance;  
    protected string name;  
}
```

- We use the access modifier `protected` to hide all fields from other classes, except derived classes.
- We use a `static` field to keep track of the assigned account numbers.

## Revisiting the BankAccount Class

```
public BankAccount(string name) {  
    latestAccountNo++;  
    this.accountNo = latestAccountNo;  
    this.name = name;  
    this.balance = 0M;  
}  
  
public BankAccount(ulong no, string name) {  
    this.accountNo = no;  
    this.name = name;  
    this.balance = 0M;  
}
```

- We use *overloading* of the constructor class and the static field to auto assign account numbers.

## Revisiting the BankAccount Class

```
public void Deposit(decimal x) {  
    this.balance += x;  
}
```

- The `Deposit` method is unchanged.
- Its access modifier is `public`.

## Revisiting the BankAccount Class

```
public virtual void Withdraw(decimal x) {
    if (this.balance >= x) {
        this.balance -= x;
    } else {
        throw new InsufficientBalance("Balance too low:
{0}", this.balance);
    }
}
```

- We use *exceptions* to cover the case of an insufficient balance for making a withdrawal.
- The method must be declared *virtual* to allow overriding in a sub-class.

## Invariants

```
// Class invariants:
// invariant: this.balance >= 0
```

- We record the above *class invariants*: this predicate must hold at any point in the lifetime of an object of this class.

## Revisiting the BankAccount Class

```
public decimal GetBalance() { return this.balance; }

public void ShowBalance() {
    Console.WriteLine("Current Balance: " + this.balance.ToString());
}

public virtual void ShowAccount() {
    Console.WriteLine("Account Number: {0}\tAccount Name: {1}\tCurrent
Balance: {2}",
        this.accountNo, this.name, this.balance.ToString());
}
```

- ShowAccount must be declared virtual to allow overriding in sub-classes.
- The other methods are unchanged.

## Revisiting the BankAccount Class

```
public class InsufficientBalance : System.Exception {
    public InsufficientBalance(string msg, decimal x):base(msg)
    {
        Console.WriteLine(" " + x.ToString());
    }
}
```

- The exception class derives from `System.Exception`
- It prints a message by calling the constructor of this base class
- Additionally, it prints the balance.

## Implementing the Class

```
class ProperBankAccount: BankAccount {  
    public decimal overdraft { get ; set;}
```

- ProperBankAccount inherits from BankAccount, thus all non-private fields and methods are available.
- The overdraft is implemented as a property with default get and set methods.

## Implementing the Class (cont'd)

```
public ProperBankAccount(string name) :base(name) {  
    // nothing; use set property on overdraft  
}
```

```
public ProperBankAccount(ulong no, string name)  
:base(no,name) {  
    // nothing; use set property on overdraft  
}
```

- We use overloading to implement 2 constructors for ProperBankAccount
- The static field is used to keep track of assigned account numbers.

## Implementing the Class

```
public override void Withdraw(decimal x) {  
    if (this.balance+this.overdraft >= x) {  
        this.balance -= x;  
    } else {  
        throw new InsufficientBalance("Balance (including overdraft) too  
low", this.balance);  
    }  
}
```

- By declaring the Withdraw method as override, the instance in ProperBankAccount overrides/replaces the one in BankAccount

## Implementing the Class

```
public override void ShowAccount() {  
    base.ShowAccount();  
    Console.WriteLine("\twith an overdraft of {0}",  
this.overdraft);  
}
```

- Similarly, the ShowAccount method is overridden to additionally show the overdraft for this account.
- base.ShowAccount ( ) calls the method in the base class.

## Implementing the Class

```
// Class invariants:  
// invariant: this.balance >= - this.overdraft
```

- Finally, we record the class invariants for this class.

Industrial Programming 17

## Testing the Class

```
public void RunTransactions(BankAccount acct) {  
    // if it has an overdraft facility, initialise its value  
    ProperBankAccount pacct = acct as ProperBankAccount;  
    if (pacct != null) {  
        pacct.overdraft = 200;  
    }  
    acct.ShowAccount();  
    acct.ShowBalance();  
    // first, deposit something  
    decimal x = 600M;  
    Console.WriteLine("Depositing " + x);  
    acct.Deposit(x);  
    acct.ShowBalance();  
    // then, try to withdraw something  
    decimal y = 400M;  
    Console.WriteLine("Withdrawing " + y);  
    try {  
        acct.Withdraw(y);  
    } catch (InsufficientBalance e) {  
        Console.WriteLine("InsufficientBalance {0} for withdrawal of {1}", acct.GetBalance(), y);  
    }  
    acct.ShowBalance();  
    // then, try to withdraw the same amount again  
    ...  
}
```

Industrial Programming 18

## The Main Method

```
public static void Main(){  
    RunTester t = new RunTester();  
  
    // create a basic account  
    BankAccount mine = new BankAccount("MyAccount");  
    // create a proper account  
    ProperBankAccount mineOvdfdt = new ProperBankAccount("MyProperAccount");  
    // collect them in an array  
    BankAccount[] accts = new BankAccount[2] { mine, mineOvdfdt };  
  
    for (int i=0; i<accts.Length; i++) {  
        t.RunTransactions(accts[i]);  
    }  
}
```

- We can use the same RunTransactions method for both accounts.
- We use polymorphism in defining the array accts, holding both types of accounts.

Industrial Programming 19

## Running the Program

```
Account Number: 1001    Account Name: MyAccount Current Balance: 0  
Current Balance: 0  
Depositing 600  
Current Balance: 600  
Withdrawing 400  
Current Balance: 200  
Withdrawing 400  
200  
InsufficientBalance 200 for withdrawal of 400  
Current Balance: 200  
Account Number: 1001    Account Name: MyAccount Current Balance: 200  
Account Number: 1002    Account Name: MyProperAccount Current Balance: 0  
with an overdraft of 200  
Current Balance: 0  
Depositing 600  
Current Balance: 600  
Withdrawing 400  
Current Balance: 200  
Withdrawing 400  
Current Balance: -200  
Account Number: 1002    Account Name: MyProperAccount Current Balance: -200  
with an overdraft of 200
```

Industrial Programming 20

# Concepts used in the Example

- *Overloading* to have several constructors with different numbers of arguments.
- *Inheritance* of methods is used to share code.
- *Overriding* of methods is used to modify the behaviour of methods in sub-classes.
- *Polymorphism* is used to collect (sub-)classes.
- *Exceptions* are used for error handling.
- *Access modifiers* are used to hide fields.
- *Properties* are used for convenience.
- *Static fields* are used to count instances.