University of Hull

Abstract classes and Interfaces

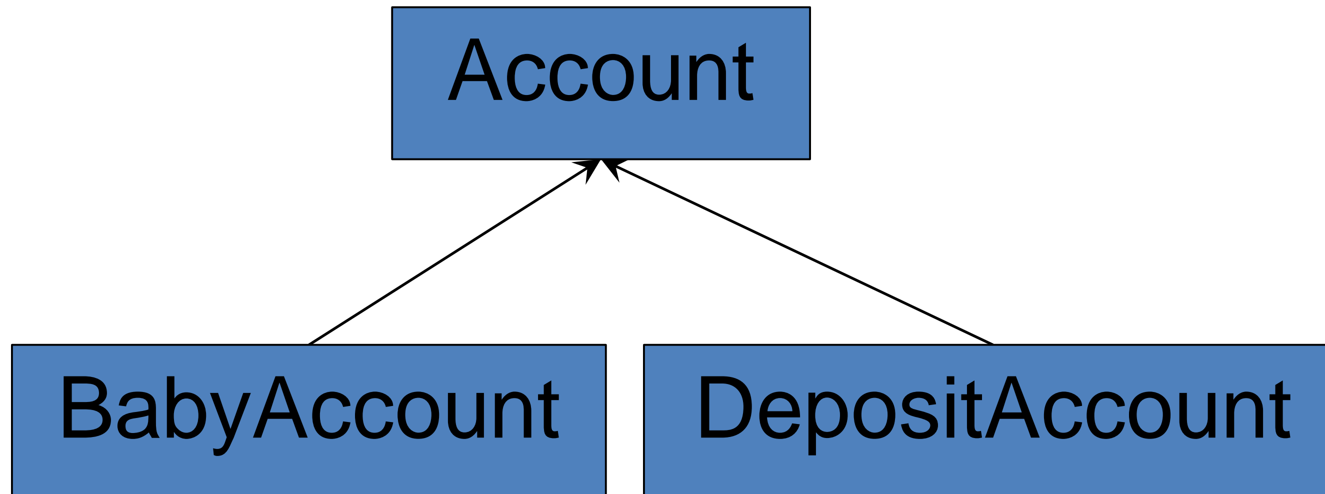
Rob Miles

Department of Computer Science

Abstraction in the Bank

- We know that there are many different kinds of accounts:
  - Current account
  - Deposit account
  - Credit card accounts
  - Baby Accounts
- A bank system must be able to handle all of these account types, plus new ones
- We have found that creating a class hierarchy is a good way to reuse code as much as possible in our solution

Accounts and Inheritance



- We know that we can make life easier with inheritance
- The child class extends the parent and adds new/different behaviours by adding methods and overriding existing ones

Abstraction

- If we "step back" from the problem we can decide that there are certain things that an account needs to be able to do:
  - Set the name of the account holder
  - Pay in funds
  - Withdraw funds
  - Read the balance
- All accounts must be able to do these things in their own way
- Exactly how they do these things will vary from one type of account, but they all need to implement the behaviours

An Abstract Account Class

```
public abstract class Account
{
    public abstract bool SetName(string NewName);

    public abstract bool PayInFunds(decimal amount);

    public abstract bool WithDrawFunds(decimal amount);

    public abstract decimal GetBalance();
}
```

- We could put all the required behaviours into an abstract class
- An abstract class is a *template* for others
- We can never make an Account instance, it just serves as the basis of other accounts

Making a Class Abstract

- When a class is made abstract you can't make an instance of it
- Instead you can extend an abstract parent class to make a child class which may not be abstract
  - We can't make an `Account`, but we can make a `CurrentAccount`
- You can make instances of the child
- It is a version of the parent template with all the methods filled in

Making a Method Abstract

- An abstract method is a placeholder
- It indicates that a child class must override this method if we want to make instances of that child class
- It does not say how the method should work, just how it is called and what it should return
- You have to make tests which ensure that the method works correctly
  - Just because a class has a method called `WithdrawFunds` doesn't mean that the method actually works

Extending an Abstract Class

- When you extend an abstract parent you must override all the abstract methods in it
  – Otherwise the child class will also be abstract
- You can think of it as filling in a template with the required methods
- The designer of the abstract class sets out the things it needs to do by specifying the abstract methods
- This means they can focus on what needs to be done, not the precise details

Class based on Account

```
public class CurrentAccount : Account
{
    public override bool SetName(string NewName)
    {
        return true;
    }

    public override bool PayInFunds(decimal amount)
    {
        return true;
    }

    // WithDrawFunds here

    // GetBalance here
}
```

Abstract Classes and Test

```
int errorCount = 0;
CurrentAccount test = new CurrentAccount ("Rob", 0);
test.PayInFunds(10);
if ( test.GetBalance() != 10)
{
    errorCount = errorCount + 1;
}
    // Lots of other tests here.....
```

- Once we have our “empty” behaviours in the `CurrentAccount` class we must create tests for them
  - Pay in 10 pounds
  - Make sure the balance increases by 10 pounds
- We can then create code that performs the tests automatically
  - If we run the above tests and `errorCount` is not 0 at the end of them we sound an alarm

Designing Abstract Parent Classes

- Not all of the methods in an abstract class need to be abstract
  - It is perfectly OK for an abstract class to contain data fields and methods
- In an `Account` all the account number management can be performed by data and code in the `Account` class
- This means that all the accounts in the hierarchy will use the same account number management code

References to Abstract Classes

- It makes sense to treat an array of bank accounts as an array of `Account` references
- An `Account` reference can refer to any of the child classes
  - A reference to a parent class can refer to instances of any of the children
- Such a reference will also be able to refer to any account class types which are created later
  - If we invent a `SuperTeenSaver` account we can add that to hierarchy and then refer to it from an array of `Account` instances

Designing Using Abstract Classes

- Step back from the problem:
  - Move from `Dress` to `StockItem`
- Identify the fundamental operations and properties
  - The ones that everyone must do in the same way are not abstract
  - The ones that have to be custom for each child class are abstract
  - Making these methods abstract forces the child class to provide its own version of that behaviour

Abstract Roundup

- Abstraction lets you consider the fundamental behaviours without worrying about individual details
- It lets you provide templates which can be filled in by specific child types
  - You can’t make instances of abstract classes
  - You can make an instance of a child which contains overrides of the abstract items
- You can set out behaviour requirements by making a class with a set of abstract methods, with one for each behaviour that you need
  - Then you create some tests for the behaviours

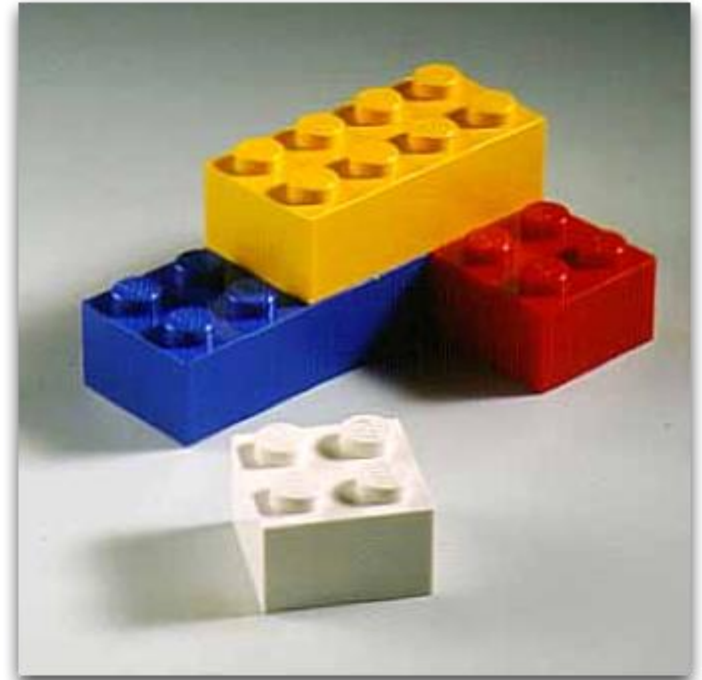
INTERFACES

Interface Introduction

- Interfaces are important
- They let programmers work with things in terms of what the things can do, not what they are
- This is a very powerful feature of C#
- It makes possible component based oriented development

Components and Interfaces

- We want to make software out of components that fit together
- This means we have to define the "plugs and sockets" that link the components
- Then we can swap components without changing



Uses for Interfaces

- There are a whole bunch of things that users of our `Account` class might want to perform
  - Add a New Account
  - Delete an Existing Account
  - Find an account
- It makes sense to be able to think of a bank in terms of these abilities, rather than a particular class
- This would make bank instances interchangeable, rather than needing to be part of a particular class hierarchy

A Bank Interface

```
interface Ibank
{
    string AddAccount(IAccount account);
    string DeleteAccount(IAccount account);
    IAccount FindAccountByName(string name);
}
```

- This is an *interface*
- An interface is a set of method headers
- By convention, the name of an interface always starts with the letter i

Implementing an Interface

```
class FriendlyBank : Ibank
{
    public string AddAccount(IAccount account)
    {
        // code that adds an account
    }

    // same for DeleteAccount and FindAccountByName
}
```

- The `FriendlyBank` class contains implementations of all the methods described in the `IBank` interface

What is the point of interfaces?

- An interface lets us manipulate something in terms of what it can do, not what it is
- In other words I can use any object that implements the `IBank` interface as a bank, and not care how it works
- This is very powerful, and adds a lot of flexibility to the design process

Bank Merger

- Consider what happens if two banks merge
- `FriendlyBank` merges with `NastyBank` to create a new bank called `StandardBank`
- Our programs must work with classes from both banks
- It can do this if the classes in the banks both implement the `IBank` interface

Interface References

```
IBank activeBank = new FriendlyBank();

...

IAccount current = activeBank.FindAccountByName("Rob");
```

- The key to understanding interfaces is understanding references to them
- The reference `activeBank` above can refer to any object that implements the `IBank` interface
  - This could be a `FriendlyBank` or `NastyBank` instance

The IAccount Interface

```
interface Ibank
{
    string AddAccount (IAccount account);
    string DeleteAccount (IAccount account);
    IAccount FindAccountByName (string name);

}
```

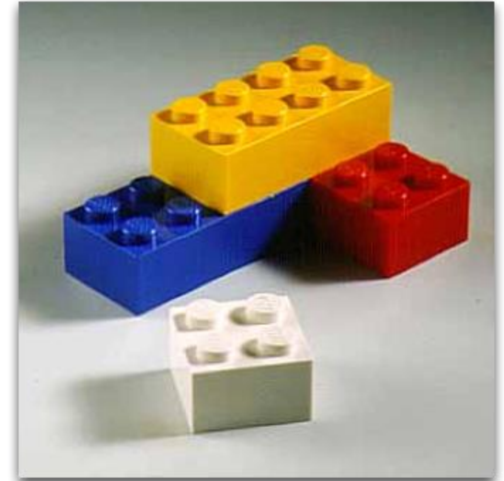
- If you look carefully at the description of the `IBank` interface you see that it uses `IAccount` references
- This means that I'm also managing by accounts in terms of interface references

`IAccount` References

- I can manage my accounts in terms of references to the `IAccount` interface
- This means that I can treat any object as an account, irrespective of which bank it came from
- The fact that it implements the interface means that it can be used as an account

Design with Interfaces

- Interfaces let you regard things in terms of what they can do
- As long as you know the object implements the behaviours you can use it in your system
- The interface is the pins on a building brick



Interface Roundup

- Interfaces decouple you from having to worry about particular classes at any point
- They let you work with things in terms of their abilities, not what they actually are
- They allow classes to bring together multiple behaviours
- They are another form of abstraction