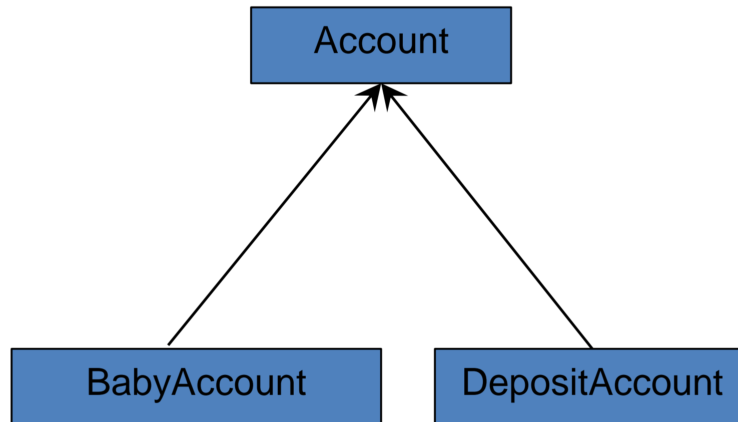
University of Hull

Design with Interfaces

Rob Miles

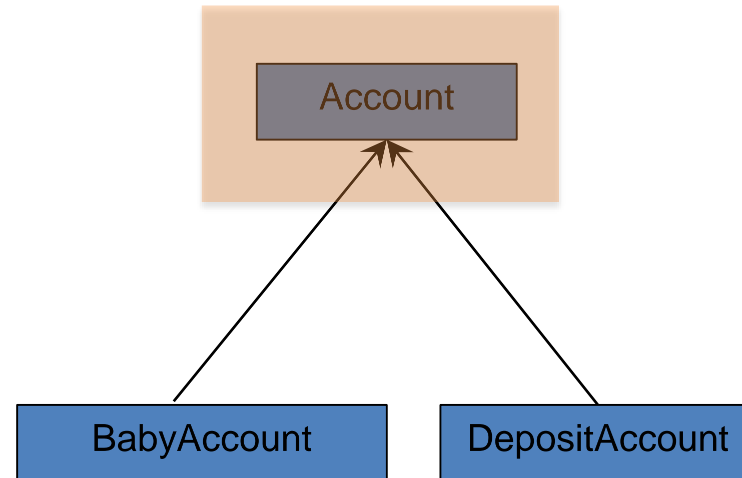
Department of Computer Science

Design and Inheritance



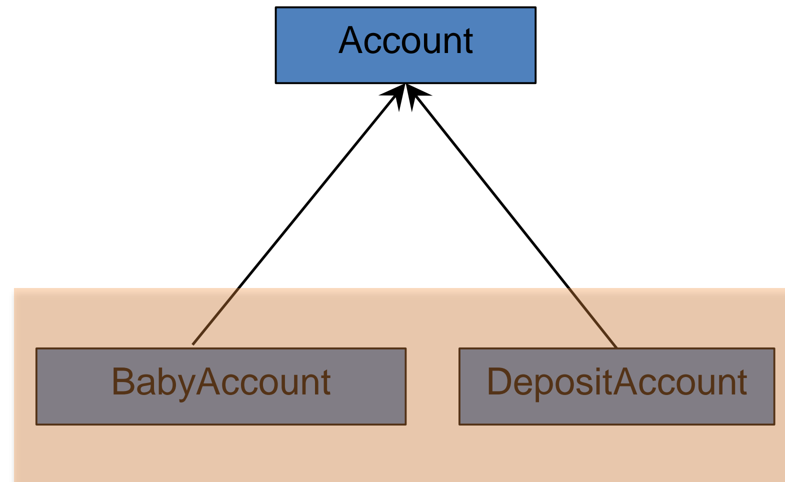
- We have seen that inheritance is a very good idea
- It lets us start with an abstraction of a component in our system and then create more specific versions of each

Design and Inheritance



- This is the parent class which is an abstraction of what an account needs to do
- We will never make an instance of the Account class because it is a template that defined “Accountness”

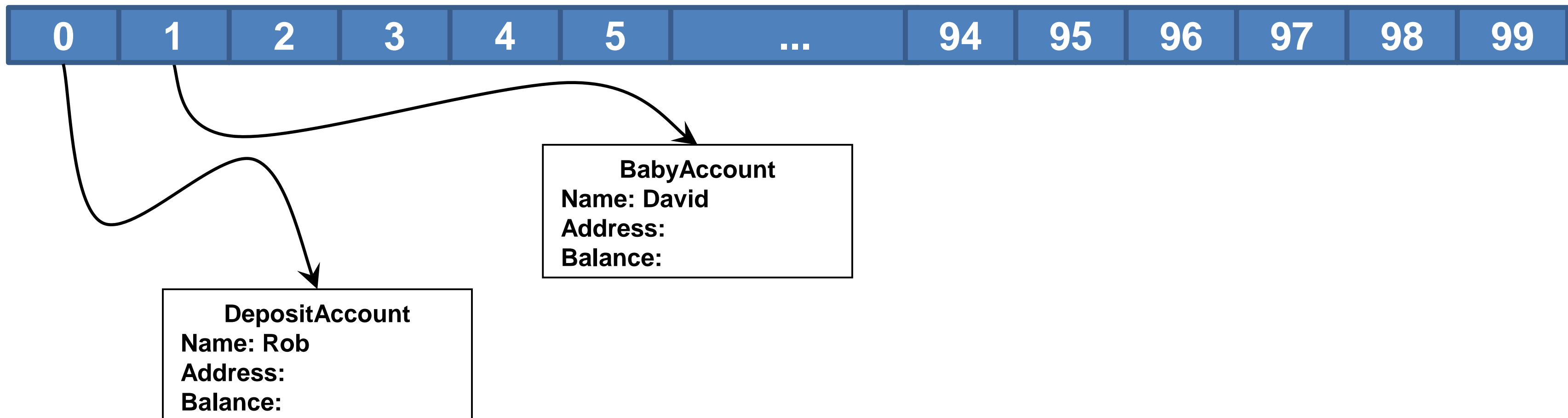
Design and Inheritance



- These are the child classes of the Abstract parent
- We will make instances of these
- They are account types that map on to specific kinds of bank customer

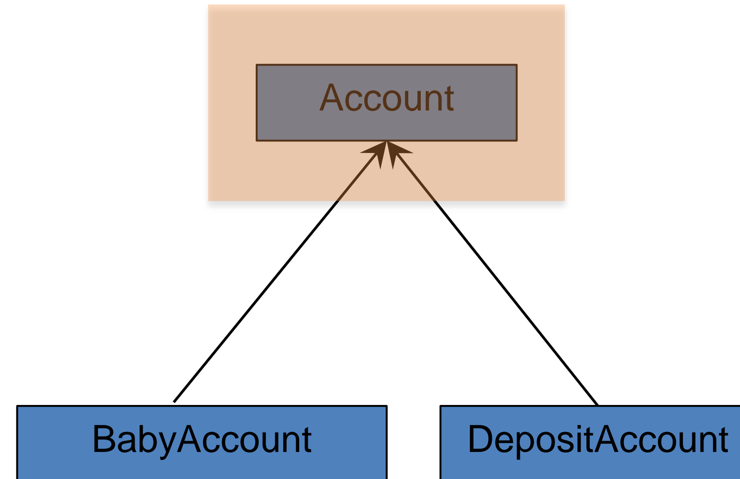
Using Abstract References

```
Account [] BankAccounts = new Account [100] ;
BankAccounts[0] = new DepositAccount("Rob");
BankAccounts[1] = new BabyAccount("David");
```



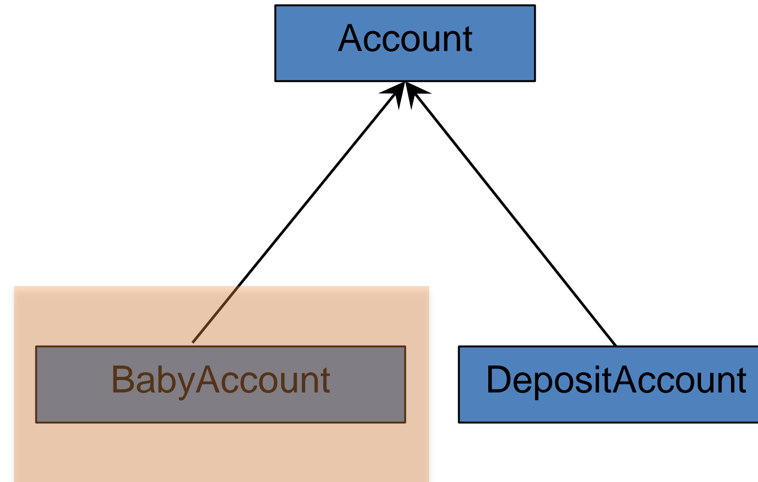
- Because a reference to a parent class can refer to an instance of any of the child classes we can store a bank as an array of `Account` references

Abstract Advantages



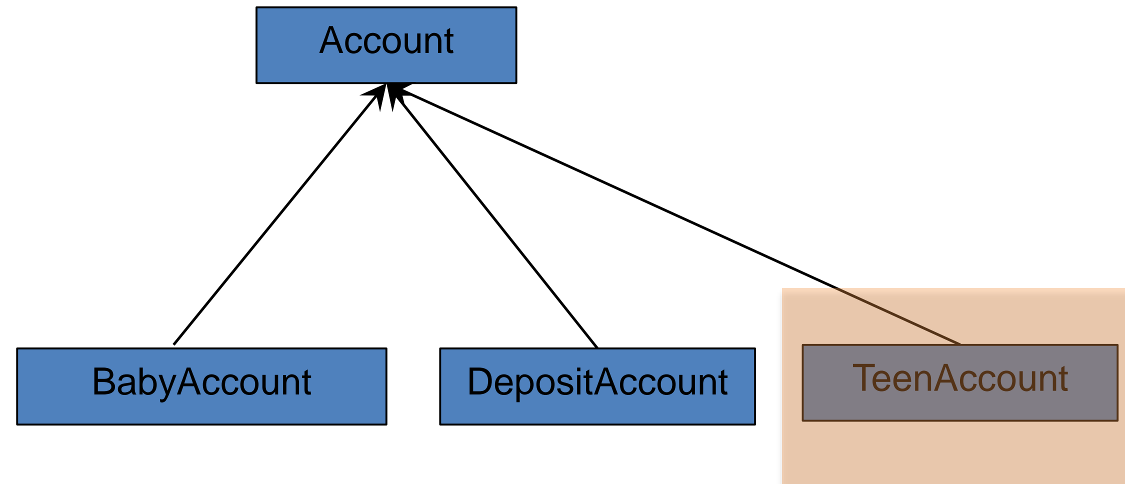
- If we need to fix a bug in the `Account` class or add a feature to it we just have to do this once in the parent class
  - All the child classes will pick up the fixed behaviour as they will use the method from the parent

Abstract Advantages



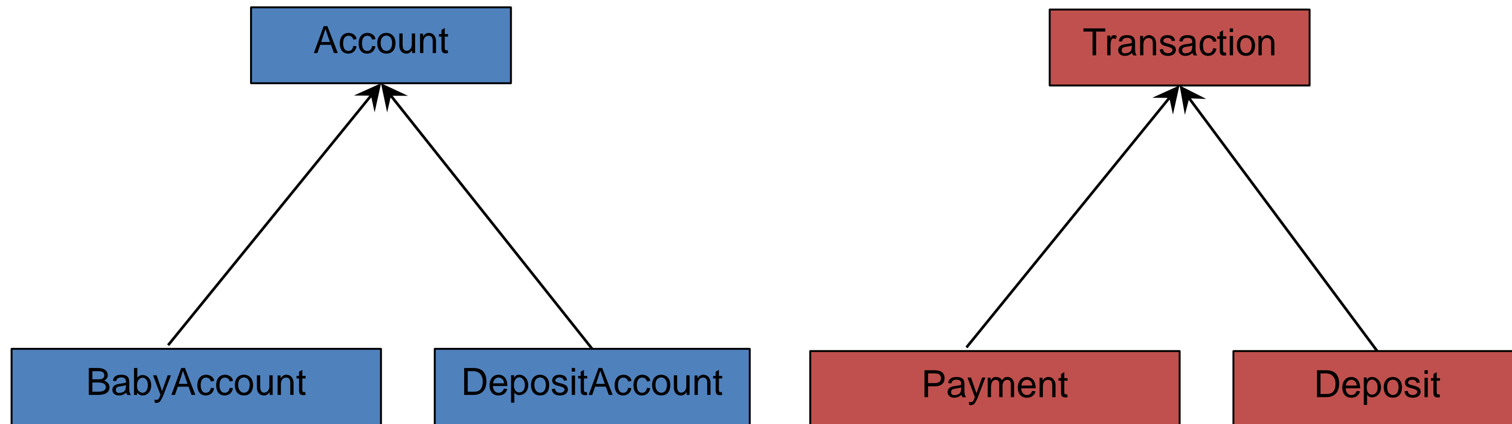
- If we need to fix a bug or add a behaviour the `BabyAccount` we just need to do it in that class
  - This change will not affect the behaviour of any of the other classes in the system

Abstract Advantages



- If a new type of account is required we can simply add it to the hierarchy in a the appropriate place – it can be “stored” in the `Account` array alongside all the existing accounts
  - We can do this as we are building the system, and even after it has been installed

Adding New Hierarchies



- In the bank we also have to keep track of transactions
  - Each transaction will be a line on a statement
- There will be many kinds of transactions, so we can use abstract classes to design this

Enter the Printer

- The bank also needs to print things out on paper
  - This includes account details and transactions
- It buys a really expensive line printer and wants you to write the software to do this

Printing Problem

- We need to make lots of objects in our bank able to be printed:
  - Accounts
  - Transactions
  - Personnel Records
  - Letters
- They are all in different class hierarchies
- We don't want the print software to have to manage separate lists of the different things that need to be printed
  - This would make it hard to add new objects later

Creating a Printing Interface

```
interface iPrint
{
    string GetPrintOutput();
    int GetLineLength();
}
```

- The best way to do this is to create in interface that describes the ability to print
- This will be used by the printer to ask an object to provide print data
- The interface could contain two methods
  - Get the message to be printed
  - Find out how many lines of text the object will need

Implementing a Printing Interface in the parent

```
abstract class Account : iPrint
{
    virtual public string GetPrintOutput()
    {
        return "Account Output\n";
    }

    virtual public int GetLineLength()
    {
        return 1; // 1 line of output
    }
}
```

- The `Account` class can implement the interface
- This means it must contain versions of the two methods

Implementing a Printing Interface in the child

```
class BabyAccount : Account, iPrint
{
    override public string GetPrintOutput()
    {
       return base.GetPrintOutput() + "BabyAccount output\n";
    }

    override public int GetLineLength()
    {
        return base.GetLineLength() + 1; // Account size plus 1
    }
}
```

- The `BabyAccount` class can also implement the interface
- This means it must also contain versions of the two methods

Overriding methods

```
class BabyAccount : Account, iPrint
{
    override public string GetPrintOutput()
    {
       return base.GetPrintOutput() + "BabyAccount output here";
    }

    override public int GetLineLength()
    {
        return base.GetLineLength() + 1;
    }
}
```

- The `GetPrintOutput` method in the `Account` class has been made `virtual`
- This means that we can `override` it in the child class

Overriding methods

```
class BabyAccount : Account
{
    override public string GetPrintOutput()
    {
        return base.GetPrintOutput() + "BabyAccount output here";
    }

    override public int GetLineLength()
    {
         return base.GetLineLength() + 1;
    }
}
```

- Inside the method we use the `base` keyboard to get the print output from the parent object
- This is **very** important

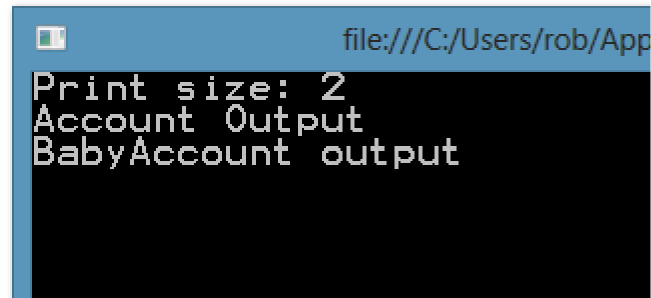
Using the Interface

```
iPrint printThing = new BabyAccount();

Console.WriteLine("Print size: " + printThing.GetLineLength());
Console.WriteLine(printThing.GetPrintOutput());
```

- I can now create a `BabyAccount` and regard it as something that implements the `iPrint` interface
- I can ask it how many lines of text it needs to print itself, and also ask it for the text to be printed
- I will get the print behaviour of the `BabyAccount`, plus the output produced by the `Account` class

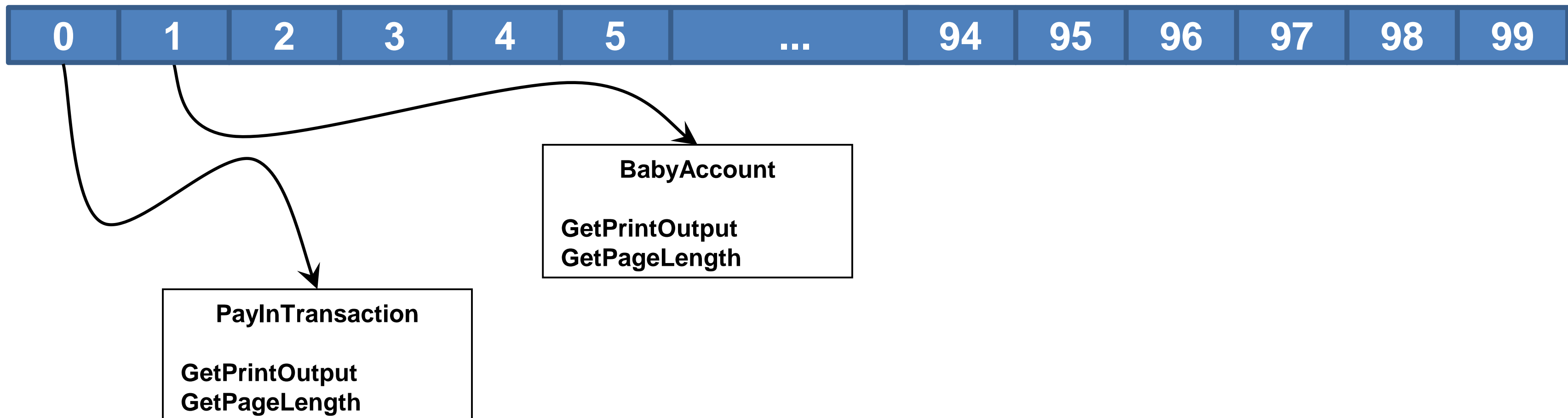
Why is this so clever?



- This is clever because if I add new stuff to the `Account` class that needs to go into the print output it will automatically appear when I print a `BabyAccount` as well
- This make maintenance of the system much easier
- You don't have to do this to use interfaces, but it is worth thinking about

Using Interface References

```
iPrint [] PrintQueue = new iPrint [100] ;
PrintQueue[0] = new PayInTransaction("Rob");
PrintQueue[1] = new BabyAccount("David");
```



- Because a reference to an interface can refer to any object that implements the interface the printer can hold a list of objects to be printed without caring what type they are

Interface methods

```
iPrint printThing = new BabyAccount();

printThing.PayInFunds(50); // This will not compile
```

- An `iPrint` reference can refer to objects that implement the `iPrint` interface
- This means that a program can only call `iPrint` methods on the reference, even if the object supports other behaviours
- This is very sensible, as the printer should never want to pay funds into an account
  - It could use casting to do this, but it might be naughty..

The problems we have just solved

- The printer can print any kind of object in our solution irrespective of their type, as long as they implement the two printing methods
  - This includes new object types that we can create after the solution has been built
- Each object can have particular print behaviours and also make use of the print behaviours of its parent
  - We can add new objects or modify the behaviour of existing ones and control precisely which parts of our system are affected

Understanding all this

- If all this seems a bit hard to understand, then don't worry
- At all times think of the problems that we are trying to solve:
  - We want to reuse code as much as possible
  - We want to make sure that we can add new objects and behaviours during the creation of the code
  - We want to ensure that objects are responsible for all their own behaviours
  - We want to make sure that objects are only manipulated in a manner appropriate to their function at that point in the program