UNIVERSITY OF Hull

Inheritance

Rob Miles

Department of Computer Science

The Problem

- The Bank Manager has decided to create a new kind of account in the bank
- It is called `BabyAccount` and it is exactly like a normal `Account`, with one difference

"Holders of BabyAccounts should not be allowed to withdraw more than 5 pounds in any transaction"

What we could to do

- One way to solve this problem would be to take a normal `Account` class and just replace the `WithDrawFunds` method
- This would mean two account types in the bank
  - We would need two different account storage arrays
  - If we needed to change the way accounts work we have to update both the `Account` and `BabyAccount` classes

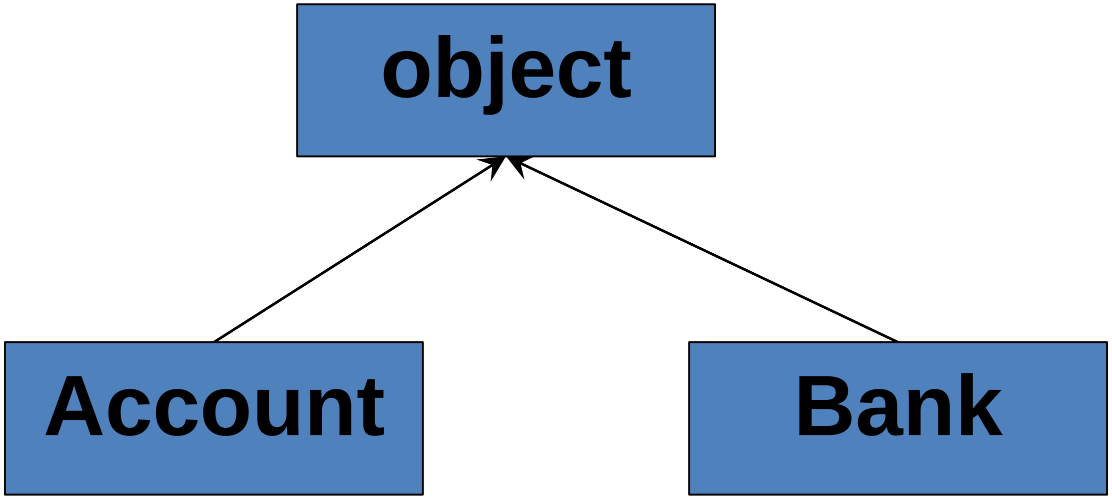
Inheritance Introduction

- Inheritance is very useful
- It lets us take existing classes and *reuse* them by *extending* them
- It can save a lot of work
  - We only have to implement the new or changed behaviour
- It is particularly valuable when writing a program to deal with lots of related items

Code Reuse With a Child Class

- I can achieve code reuse by extending a parent class and making a child class
- The child can do everything that the parent class can do
- We can add new methods to the child, or even *override* the ones in the parent
- This is the basis of inheritance

Inheritance So Far



- Whenever we create a new class it is actually an extension of the object class
- This means that all the classes we have created so far have been based on object

Object Methods

- The `object` class contains a number of methods:
  - `ToString` – returns a string description of the object contents
  - `Equals` – used to compare two instances and return true if the content of the two is the same
  - `GetHashCode` – gets a hash value for an instance
    - The hash value is a (hopefully) unique value for an instance that can be used to identify it
- Every child of `object` can do these things
  - But they often provide their own custom versions by overriding the ones in the object class

Overriding the ToString method

```
public override string ToString()
{
  return "Account: " + accountNumber +
         " Name: " + name +
         " Address: " + address +
         " Balance: " + balance;
}
```

- This version of `ToString` returns a string that describes the content of an `Account`
- It *overrides* the `ToString` method in `object`

Overriding Methods

- Overriding is where you provide a new version of a method in a child class
- The new method *overrides* the one in the parent
- It must have the same name and signature as the one in the parent
- This is **not** the same as *overloading*
  - Overloading is where the same method name is used with a variety of different method signatures

A Simple Account

```
class Account
{
    private decimal balance = 10;
    public virtual bool WithdrawFunds(decimal amount)
    {
        if (amount < balance)
        {
            balance = balance - amount;
            return true;
        }
        return false;
    }
}
```

- This is a very simple class which has a fixed amount in the bank and a single `WithdrawFunds` method

Using the Account

```
if (a.WithdrawFunds(6))
{
    Console.WriteLine("Withdraw succeeded");
}
```

- We can create account instances and then withdraw funds
- The above code would work as the `Account` is created with 10 pounds already in it
- We could add all the other methods to make a complete `Account` class

Making a BabyAccount

- A `BabyAccount` class must be able to do all the things that the parent class can do
- The only difference is in the behaviour of the `WithDrawFunds` method
- We can do this by creating a `BabyAccount` class which is a *child* of the `Account` class
- We then override the `WithDrawFunds` method in the child class

A BabyAccount class

```
class BabyAccount : Account
{
    public override bool WithdrawFunds(decimal amount)
    {
        if (amount > 5)
        {
            return false;
        }
        return base.WithdrawFunds(amount);
    }
}
```

- The header of the class states that it extends the `Account` class
- The parent class name follows the colon

A BabyAccount class

```
class BabyAccount : Account
{
    public override bool WithdrawFunds(decimal amount)
    {
        if (amount > 5)
        {
            return false;
        }
        return base.WithdrawFunds(amount);
    }
}
```

- The `WithdrawFunds` method overrides the `WithDrawFunds` in the parent class
- This method must have been made *virtual*

A Virtual Method

```
class Account
{
    private decimal balance = 10;
    public virtual bool WithdrawFunds(decimal amount)
    {
        if (amount < balance)
        {
            balance = balance - amount;
            return true;
        }
        return false;
    }
}
```

- Only methods marked as `virtual` can be overridden
- The compiler must generate different code to call a method that might be overridden

BabyAccount WithdrawFunds

```
class BabyAccount : Account
{
    public override bool WithdrawFunds(decimal amount)
    {
        if (amount > 5)
        {
            return false;
        }
        return base.WithdrawFunds(amount);
    }
}
```

- The method refuses to let the baby withdraw more than 5 pounds
- If the amount is less than this limit the `WithdrawFunds` method in the parent class is called to do the withdrawal

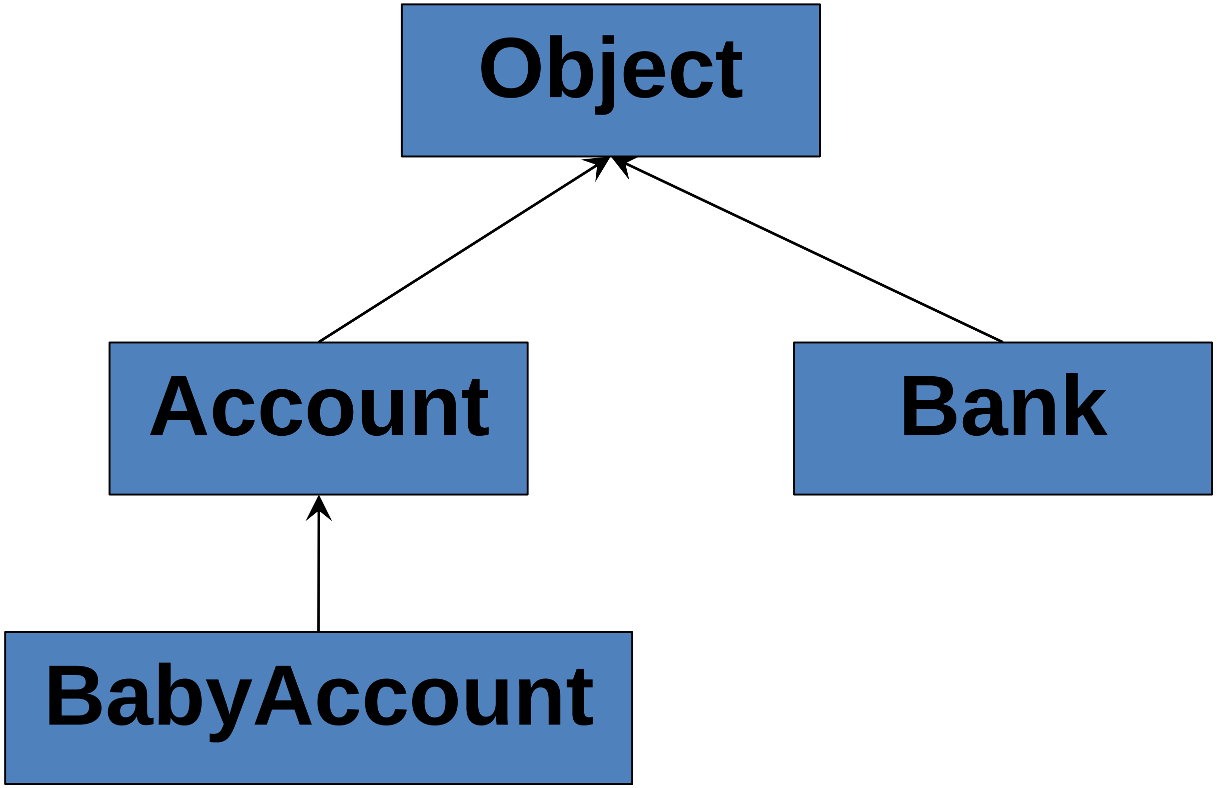
Using the base keyword

```
public override bool WithdrawFunds(decimal amount)
 {
     if (amount > 5)
     {
         return false;
     }
     return base.WithdrawFunds(amount);
 }
}
```

- Putting `base.` in front of the method causes the one in the parent class to be called

- This is sensible, since the `Account` can then update the balance (which the `BabyAccount` does not have access to)

Bank Class Diagram



- The lower down the hierarchy, the **more** a class can do

Understanding Hierarchies

- It is important that you remember that the child can always do at least as much as the parent
- It can do more if it contains extra data and methods:
  - The `BabyAccount` could contain the name and address of the guardian of the baby
- You can also regard classes further down the hierarchy as more specialised
  - The ones at the top are general, then the ones further down are for specific situations

Using a BabyAcount

```
BabyAccount b = new BabyAccount();
b.PayInFunds(100);
if (b.WithdrawFunds(4))
{
    Console.WriteLine("Withdraw succeeded");
}
```

- The `PayInFunds` method in the `Account` class is used to pay money in, since it has not been overridden in the `BabyAccount`
- However, the `WithdrawFunds` method in the `BabyAccount` class will be used when money is withdrawn

Overriding Considerations

- When you call a method on an instance of a child class the run time system searches up the class hierarchy for that method, starting at the child
- The first method that is found is called
- The `base` keyword causes a search for the next method "above" this one
- Child classes needn't override all the methods in the parent
  - You should only override the methods that you need to

Overriding in Class Design

- When you design your classes you only make methods virtual if you know that they may need to be overridden
- It is unlikely that we would override the `GetBalance` method, so this would not be virtual
- However, `PayInFunds` might need to be overridden
  - there may be accounts where we want to limit the amount of cash paid in with a single transaction
  - `PayInFunds` should be made `virtual` to allow this

Child and Parent Construction

- A child instance is constructed based on a parent
- You can't have a child without a parent
- In other words, to make a `BabyAccount` we must first make an instance of an `Account`
- This has ramifications for the construction process
  - Especially if the parent class has a constructor which must be called to create an instance of the parent

Adding a Constructor to Account

```
public Account(decimal initialBalance)
{
    balance = initialBalance;
}
```

- We could use a constructor to our simple account which sets the initial balance
  - In fact we have much more complex constructors in the real Bank application
- Unfortunately this breaks our program:

```
"No overload for method 'Account' takes '0' arguments"
```

Constructor Chaining

```
public BabyAccount(decimal initialBalance)
    : base(initialBalance)
{
}
```

- The constructor for `BabyAccount` must call a constructor in the parent class to make the parent instance
- The `base` keyword is used to achieve this
- It makes a call to a constructor in the parent class
- That way an `Account` is made before the `BabyAccount`

Constructing Constructors

- It is important that when you create your classes you consider how each class will be constructed
- The constructor at each level must call one in the parent before setting the values at that level in the hierarchy
- This is an important aspect of the class design process

References in Class Hierarchies

- Classes are managed by reference
  - We create tags which refer to an object instance in memory
- The C# compiler is very strict about reference types
  - It ensures that object references are *typesafe*
- This has implications when we use references in class hierarchies

Child Classes and References

- Classes are managed by reference
  - We create tags which refer to an object instance in memory
- There is a fundamental principle in class hierarchies:

The Child can always do more than the Parent

- Every time you add a layer you pick up all the behaviours of the layer above
- This has implications when we consider references

Parent and Child References

- It is permissible for a reference to a parent class to refer to an instance of a child
  - This is because the child can always do everything the parent can do

```
BabyAccount babyRef = new BabyAccount(100);
b.WithdrawFunds(4);
Account accountRef = babyRef;
accountRef.WithdrawFunds(1);
```

- This code will work fine, `accountRef` and `babyRef` both refer to the same `BabyAccount` instance and the `BabyAccount` instance has a behaviour for every `Account` behaviour

Child and Parent References

- It is impossible for a reference to a child class to refer to an instance of a parent
  - This is because the parent cannot always do what the child can
- If the child has additional behaviours, these are not present in the parent
- The compiler will complain if you try to do this
- To see what we mean, here is an example….

Storing Parent Names in BabyAccount Instances

```
class BabyAccount : Account
{
    string parentName;
    public string GetParent()
    {
        return parentName;
    }
}
```

- The `BabyAccount` could contain the name of the parent of the account holder
- It would have a method called `GetParent` to get this name value

Using the GetParent method

- It is not permissible for a reference to a child class to refer to an instance of a parent:

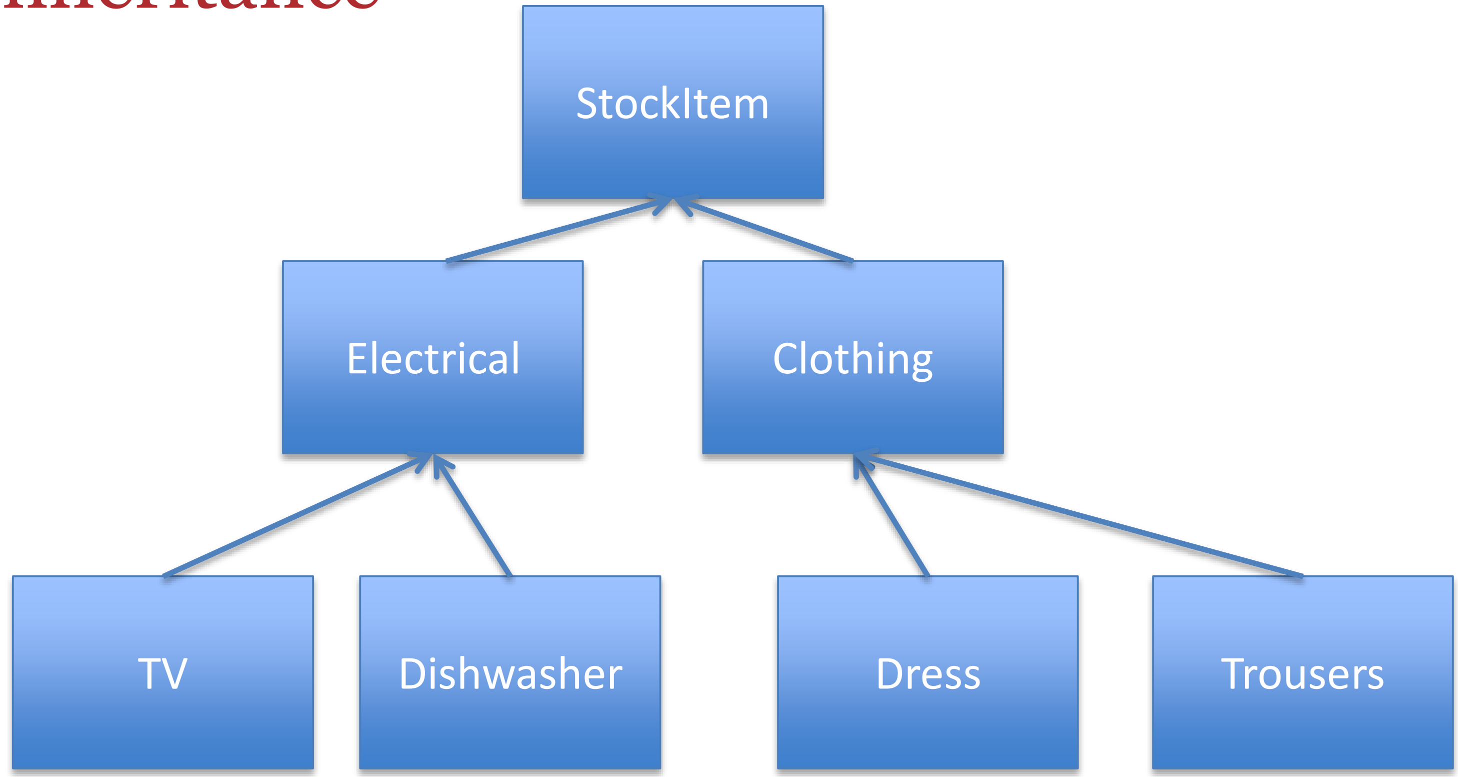
```
Account accountRef = new Account(100);
BabyAccount babyRef = accountRef; // This will not compile
```

- This code will work not compile
- This is because the `Account` class does not have a `GetParent` behaviour, which the `babyRef` is expecting
- The compiler makes sure that the object on the end of a reference can do all the things the reference needs

Reference Power

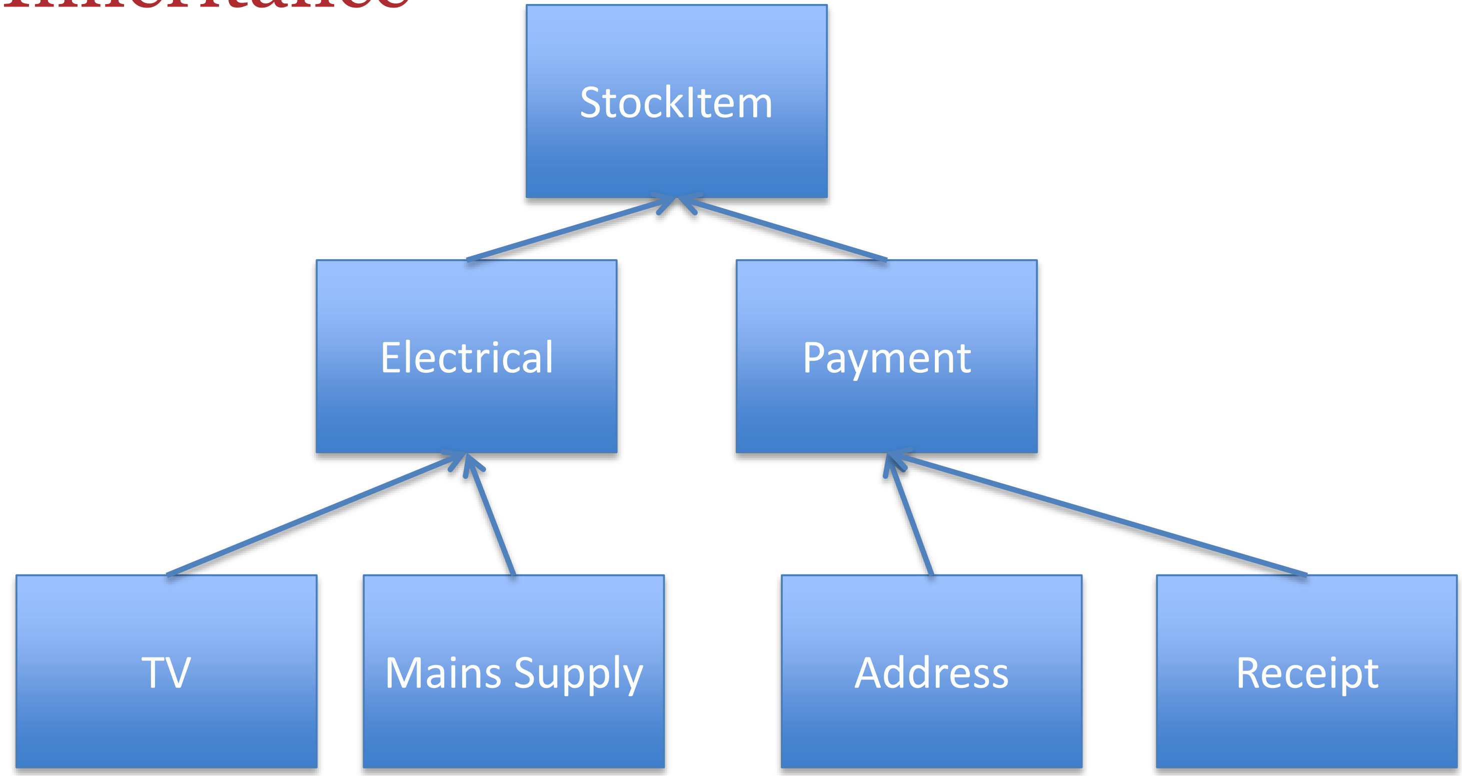
- The real power of references in hierarchies is that since a reference to a parent can refer to any of the children we can still use an `Account` array to keep track of `BabyAccounts`
- We can even override the `Load` and `Save` methods in the `BabyAccount` class so that they behave correctly
- And we can add new account types as required by the customer

Using Inheritance



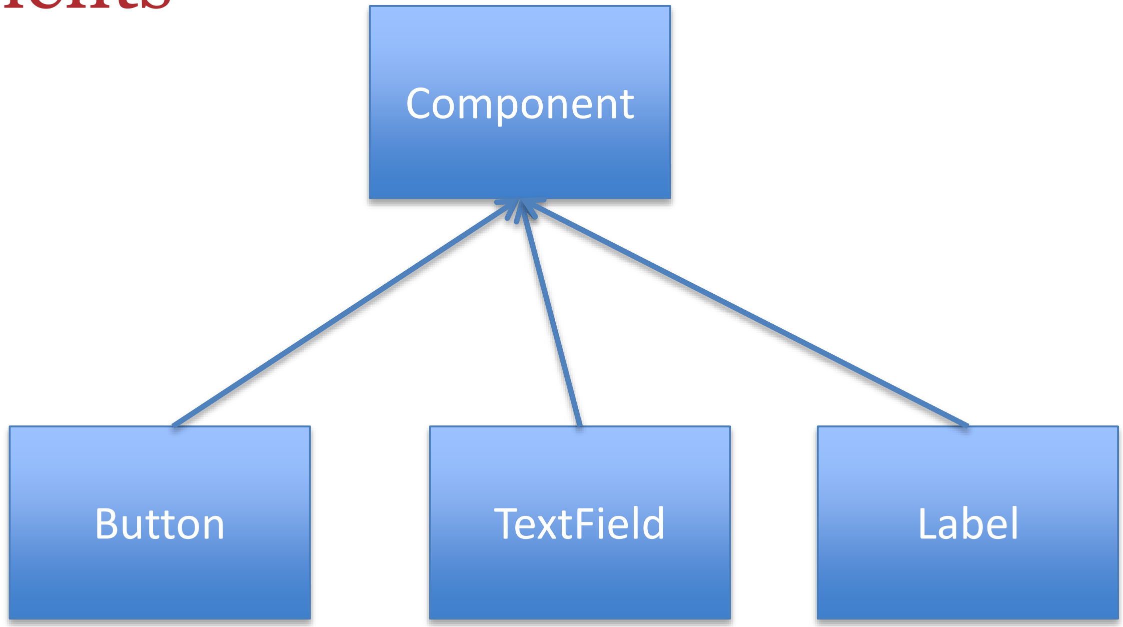
- Inheritance lets us customise code to make objects that reflect more specialised requirement
- It also allows us to extend an existing system in the light of future requirements

Stupid Inheritance



- It is important that all the items in the inheritance tree are part of a “family” of related items

Components



- The Windows Forms components are based on a class hierarchy
- You can create your own versions of the components by extending these component classes yourself

Sensible Inheritance

- Make sure that all the classes are related
  - Everything in the hierarchy should be a version of the item at the top
- Don't make the class hierarchy too deep
  - This makes things complicated and can slow programs down
- Make sure the top class is abstract enough
  - The top class in a dress shop should be `StockItem` not `ClothingItem`, so the shop can sell handbags…

Inheritance and Components

- Inheritance is not a magic bullet
  - It doesn’t solve all your problems, it simply makes it easier to reuse code in some situations
- Inheritance is particularly useful when you are creating a set of related resources
  - The WPF elements are all part of a class hierarchy
  - Each element further down the hierarchy adds an additional behaviour or works slightly differently
- Modern program design makes use of interfaces to generate interchangeable components

Inheritance Review

- A class can extend a parent class
  - This means it has the same data and methods as the parent
- Methods can be marked as *virtual* so that they can be *overidden* by code in the child class
  - This lets us create child classes with customised behaviours
- A reference to the parent class can refer to any of the child classes
  - But it can only use the behaviours in the parent