


Awarding
Great British
Qualifications




Designing and Developing Object-Oriented Computer Programmes
Topic 8:
Object Orientation (2)

1

Scope and Coverage

This topic will cover:

- *Objects Revisited*
- *Inheritance*
- *Making good use of inheritance*
- *Polymorphism*
- *AdminUser extensions*




2

Learning Outcomes

By the end of this topic students will be able to:

- *Make use of inheritance;*
- *Understand and apply polymorphism;*
- *Override methods through the use of virtual keywords*




3

Title of Topic: Topic 1 - 1.4

Objects Revisited - 1

- We saw in Topic 5 how useful objects could be for grouping together sets of related data into a single unit we could reference throughout the program.
- We discussed how we could **encapsulate** data and the methods that act upon that data together.
- We saw in the last chapter how we could make lists of objects through the use of array type data structures and create our own simple databases.




4

Title of Topic: Topic 1 - 1.5

Objects Revisited - 2

- In this chapter, we're going to discuss the real reasons why objects are so intensely powerful – the role of **inheritance** and the flexibility offered by **polymorphism**.
- These, together with encapsulation, represent the three key pillars of Object Orientation, and effective use of this programming style depends on being comfortable with all of them.




5

Title of Topic: Topic 1 - 1.6

Inheritance - 1

- Often when building an object oriented program we will encounter a scenario when we need to duplicate large amounts of functionality within two objects.
- Consider our cash machine scenario – we have only a single kind of account, but most banks offer many kinds of accounts with many different features.




6

Title of Topic: Topic 1 - 1.7

Inheritance - 2

- Some offer overdrafts.
- Some have a minimum deposit amount per month.
- Some offer only delayed withdrawals.
- Some have both an authorised and unauthorised overdraft facility.
- In all cases, they'll share a certain set of attributes – the owner of the account, and the balance.
- Some though will have more things they track, and some will treat the common things differently.




7

Title of Topic: Topic 1 - 1.8

Inheritance - 3

- Throughout the course of this unit we've been looking at the ways we can reduce or eliminate code duplication.
- **Arrays** permit us to cut down on the number of variables we use.
- **Loops** allow us to have the program handle repetition rather than the programmer.




8

Title of Topic: Topic 1 - 1.9

Inheritance - 4

- However, with **objects** we've introduced the problem once more – we're now in the situation that if we want objects that do similar but different things, we have to duplicate all the code.




9

Title of Topic Topic 1 - 1.10

Inheritance - 5

- Or rather, we **would** have to duplicate were it not for inheritance.
- Inheritance permits us to set a class as the **parent** of another class.
- The **child** of that class takes on all the attributes and methods of the parent, and can provide additional functionality either through **extension** (the provision of more attributes and methods) or **specialisation** (changing how existing methods work).
- Most child classes will make use of a combination of these approaches.




10

Title of Topic Topic 1 - 1.11

Inheritance - 6

- Let's look at a simple example – a User class that serves as the **parent** to both a NormalUser and an AdminUser.
- For both of the child classes we'll want to store a real name, a user name, and a password.
- The class definition would be as follows:



11

Title of Topic Topic 1 - 1.12


Inheritance - 7

```
class User
{
    private String realname;
    private String username;
    private String password;

    public String Realname
    {
        get
        {
            return realname;
        }
        set
        {
            realname = value;
        }
    }
}

public String Username
{
    get
    {
        return username;
    }
    set
    {
        username = value;
    }
}

public String Password
{
    get
    {
        return password;
    }
    set
    {
        password = value;
    }
}
```




12

Title of Topic: Topic 1 - 1.13

Inheritance - 8

- To create our NormalUser, we don't need to duplicate any of these.
- We simply create a new class as normal, and as part of our class definition indicate our desire to **inherit** the methods and attributes of a parent:

```
class NormalUser : User
{
}
```




13

Title of Topic: Topic 1 - 1.14

Inheritance - 9

- The colon there indicates that we're specifying an inherited relationship.
- The name that follows it is the class from which we are inheriting.
- You've seen this notation before – it's what's there in all the forms we create:

```
public partial class frmInheritance : Form
```




14

Title of Topic: Topic 1 - 1.15

Inheritance - 10

- This is why we don't need to do anything about telling C# what a Form is or how it should work.
- There's a Form class already written, and we're saying 'one of those, please'.
- When we add event handlers and the like we are **extending** the functionality of the Form class.




15

Title of Topic: Topic 1 - 1.16

Inheritance - 11

- Right from the start, our NormalUser now has all the methods and attributes of its parent class.
- If we added nothing else to our definition, it would function exactly as a User class.
- You can see that easily enough by attempting to access its properties:

```
NormalUser u;  
  
u = new NormalUser();  
  
u.Username = "mjh";
```



16


Title of Topic: Topic 1 - 1.17

Inheritance - 12

- Although we didn't add any of the attributes or the properties, they're declared in the parent class so we get them for free in the child.
- Similarly if we create an AdminUser:

```
class AdminUser : User  
{  
}  

```



17


Title of Topic: Topic 1 - 1.18

Inheritance - 13

- And then attempt to make use of its properties:

```
NormalUser u;  
AdminUser a;  
  
u = new NormalUser();  
a = new AdminUser();  
  
u.Username = "mjh";  
a.Username = "admin1";
```

- That's interesting, but it's not useful yet. Let's look at making these objects do something worthwhile.




18

Title of Topic: Topic 1 - 1.19

Making good use of Inheritance - 1

- We're going to differentiate these classes based on how many times we can fail to logon before we get locked out.
- If we fail to enter the right password for an admin account three times, the account locks.
- A normal user never gets locked out.




19

Title of Topic: Topic 1 - 1.20

Making good use of Inheritance - 2

- That means the following:
 - Both accounts are going to need some code to check if a password is correct
 - The admin account will need to track how many unsuccessful logins there have been, and whether the account is locked.
 - The admin account only will refuse to validate a login if it is locked



20

Title of Topic: Topic 1 - 1.21


Making good use of Inheritance - 3

- The first of these is shared functionality, so this is where inheritance starts to become useful – we don't write the code for this in either child class.
- We put it into the parent so they can share it:

```
class User
{
    private String realname;
    private String username;
    private String password;

    // Property setup as usual

    public Boolean checkPassword(string pass)
    {
        if (pass.ToLower().Equals(password.ToLower()))
        {
            return true;
        }
        return false;
    }
}
```




21

Title of Topic: Topic 1 - 1.22

Making good use of Inheritance - 4

- Just like that, both NormalUser and AdminUser have a checkPassword method available.
- Let's try it out, making use of a simple login application.
- We're going to create a list of our NormalUser objects, and then provide the appropriate 'access granted' or 'access denied' message when the login button is pressed.




22

Title of Topic: Topic 1 - 1.23

Making good use of Inheritance - 5

- We'll populate this to begin with three accounts:

```
public partial class frmInheritance : Form
{
    List<NormalUser> accounts;
    public frmInheritance()
    {
        InitializeComponent();
    }
    private void addAccount(String name, string user, string pass)
    {
        NormalUser u;
        u = new NormalUser();
        u.RealName = name;
        u.Username = user;
        u.Password = pass;
        accounts.Add(u);
    }
    private void frmInheritance_Load(object sender, EventArgs e)
    {
        accounts = new List<NormalUser>();
        addAccount("Michael Heron", "mjh", "bing1");
        addAccount("Alan Moon", "ami", "ticket");
        addAccount("Aoife Lockhart", "awl", "euro");
    }
}
```




23

Title of Topic: Topic 1 - 1.24

Making good use of Inheritance - 6

- Our job now is going to be to find the account the user enters, and then check the provided password.
- First, to find an account:

```
private NormalUser FindAccount(string us) {
    for (int i = 0; i < accounts.Count; i++)
    {
        if (accounts[i].Username.ToLower() == us.ToLower())
        {
            return accounts[i];
        }
    }
    return null;
}
```




24

Title of Topic Topic 1 - 1.25

Making good use of Inheritance - 7

- Here we step over each account in our list, checking to see if the lower case of the username matches the lower case of the string we provided in the username textbox.
- If it does, we return the account object found in the current position of the list.
- Returning null means we didn't find anything.



25

Title of Topic Topic 1 - 1.26

Making good use of Inheritance - 8


- Our login code at the end looks like this:

```
private void cmdLogin_Click(object sender, EventArgs e)
{
    NormalUser user;

    user = findAccount(txtLogin.Text);

    if (user == null)
    {
        MessageBox.Show("No such account.");
        return;
    }

    if (user.checkPassword(txtPassword.Text))
    {
        MessageBox.Show("Access granted.");
    }
    else
    {
        MessageBox.Show("Access Denied.");
    }
}
```




26

Title of Topic Topic 1 - 1.27

Making good use of Inheritance - 9

- When we run this program we find that we get the appropriate messages based on our login.
- If we replaced every mention of NormalUser with AdminUser, we'd find exactly the same thing.
- At the moment, our classes are identical.
- That's about to change.




27

Title of Topic: Topic 1 - 1.28

Polymorphism - 1

- You might have noticed a problem here – if we are using a typed list, how can we have accounts of different types stored within?
- We could of course create a list of normal users and a list of admin users, but then we need to be constantly checking through two different lists and that seems like more work than it's worth.
- Here's where the next powerful object oriented technique comes into play – **polymorphism**.




28

Title of Topic: Topic 1 - 1.29

Polymorphism - 2

- Polymorphism is the code convention of treating instances of a specialised object (a child) as instances of a more general object (one of its parents).
- Our NormalUser is a NormalUser, but it's also a specialised type of User.
- Within object oriented programming, we can treat it as either case.




29

Title of Topic: Topic 1 - 1.30

Polymorphism - 3

- That doesn't sound useful until we realise that AdminUser is an AdminUser, but it is **also** a specialised kind of User.
- If we can treat both NormalUser **and** AdminUser as a type of User, then we have a solution to our problem.
- When we provide type information, we provide it using the most general form the object will take.
- In this case, User.



30

Title of Topic: Topic 1 - 1.31

Polymorphism - 4

- Our program above, if written polymorphically, would look like this:

```
public partial class frmInheritance : Form
{
    List<User> accounts;
    public frmInheritance()
    {
        InitializeComponent();
    }
    private User findAccount(string us) {
        for (int i = 0; i < accounts.Count; i++)
        {
            if (accounts[i].Username.ToLower() == us.ToLower())
            {
                return accounts[i];
            }
        }
        return null;
    }
    private void addAccount(string name, string user, string pass)
    {
        NormalUser u;
        u = new NormalUser();
        u.RealName = name;
        u.Username = user;
        u.Password = pass;
        accounts.Add(u);
    }
    private void frmInheritance_Load(object sender, EventArgs e)
    {
        accounts = new List<User>();
        addAccount("Michael Heron", "mjh", "bing1");
        addAccount("Alan Moon", "am", "tictac");
        addAccount("Noife Lockhart", "nli", "euro");
    }
    private void cmdLogin_Click(object sender, EventArgs e)
    {
        User user;
        user = findAccount(txtLogin.Text);
        if (user == null)
        {
            MessageBox.Show("No such account.");
            return;
        }
        if (user.checkPassword(txtPassword.Text))
        {
            MessageBox.Show("Access granted.");
        }
        else
        {
            MessageBox.Show("Access Denied.");
        }
    }
}
```

NCC

31

Title of Topic: Topic 1 - 1.32

Polymorphism - 5

- Now, we need to look at what's actually happening here because it looks like we're randomly choosing when to use NormalUser and User.
- When we add an account, we create a NormalUser variable as usual, but when we use findAccount we are asking only for a User.
- This is perhaps the most difficult thing to become comfortable with in Polymorphism, and it's to do with the **contract** our program forms with the underlying virtual machine.

NCC

32

Title of Topic: Topic 1 - 1.33

Polymorphism - 6

- When we provide code that makes use of methods or variables that don't exist, the compiler gives up on the job of trying to turn it into a working program.
- This is because we are asking it to do something it can't do.
- In many cases, that's obvious at the time because key information is missing.


NCC

33

Title of Topic: Topic 1 - 1.34

Polymorphism - 7

- With polymorphism though, we may not know that there's missing information until the program is already running.
- When the compiler lets us do something, what it's essentially saying is 'Okay, when the virtual machine gets to this I'm confident that it'll have the information it needs'.




34

Title of Topic: Topic 1 - 1.35

Polymorphism - 8

- When we create a user, we use NormalUser because we know that's the kind of user we want.
- However, the compiler doesn't know in what order we'll add things to our list.
- So we tell it to treat everything on the list as a User, but what we add is the **specialised** object that we actually want.




35

Title of Topic: Topic 1 - 1.36

Polymorphism - 9

- We don't lose any information with Polymorphism – the object we add to the list isn't a User, it's a NormalUser with all the specialisations and extensions that implies.
- It's just that under most circumstances the virtual machine is going to treat it in a more general sense.



36

Title of Topic: Topic 1 - 1.37

Polymorphism - 10

- When it is time for the virtual machine to execute the FindAccount method, it thinks 'okay, everything in this list is **at least** of the level of being a User.
- That means that anything that's been defined as part of the User class is available to me'.
- That's the contract we're forming with polymorphism – that when we use the more general case we're going to restrict ourselves to the functionality that is guaranteed to be there by the parent class structure.

NCC

37

Title of Topic: Topic 1 - 1.38

Polymorphism - 11

- To make our AddAccount method work polymorphically, we'd do this:

```
private void addAccount(String name, string user, string pass, Boolean admin)
{
    User u;

    if (admin == true)
    {
        u = new AdminUser();
    }
    else
    {
        u = new NormalUser();
    }

    u.RealName = name;
    u.Username = user;
    u.Password = pass;

    accounts.Add(u);
}
```

NCC

38

Title of Topic: Topic 1 - 1.39

Polymorphism - 12

- Here we're saying 'u is going to be a User of some kind, but I'm not sure what kind until the function is called'.
- We put the appropriate specialised object into the polymorphic User object, and everything works the way we'd expect.
- Again, we'll come back to that in a little bit.

NCC

39

Title of Topic: Topic 1 - 1.40

Polymorphism - 13

- But now, let's adjust our code a little to take into account the new AdminUser objects we're using:

```
addAccount("Michael Heron", "mjh", "bing1", true);
addAccount("Alan Moon", "am1", "ticket", false);
addAccount("Aoife Lockhart", "awl", "euro", true);
```

- Now mjh and awl are admin, but am1 is not. Let's make that actually **mean** something.

NCC

40

Title of Topic: Topic 1 - 1.41

AdminUser extensions - 1

- Now we need to set the admin account as being potentially locked, and also keep track of the number of failed logins.
- The attributes are the easiest to do, so let's add them:

NCC

41

Title of Topic: Topic 1 - 1.42

AdminUser extensions - 2

```
class AdminUser : User
{
    private int failedLogins;
    private Boolean locked;

    public int FailedLogins
    {
        get
        {
            return failedLogins;
        }

        set
        {
            failedLogins = value;
        }
    }

    public bool Locked
    {
        get
        {
            return locked;
        }

        set
        {
            locked = value;
        }
    }
}
```


NCC

42

Title of Topic: Topic 1 - 1.43

AdminUser extensions - 3

- Here's one of the areas where polymorphism has an impact – whenever we're dealing with the more general class, we can't use any properties, attributes or methods that are defined in a more specialised one.
- We can't have our findAccount method for example make use of the FailedLogins property because there is no way to guarantee that it's available in every child of the User class.




43

Title of Topic: Topic 1 - 1.44

AdminUser extensions - 4

- NormalUser, as an example, doesn't have these.
- If we want to get access to these values, we need to be doing it either **within** the class itself, or with a specialised object that is typed appropriately.




44

Title of Topic: Topic 1 - 1.45

AdminUser extensions - 5

- We don't have the latter of those, but what we can do is **specialise** the behaviour of our login code within the class.
- For this, we use a process called **overriding**.
- We discussed **overloading** earlier in this unit – that's the technique of providing two methods with the same name but different types of parameters.
- Overriding is when we take a method name and provide a more specialised version of it in a child class.




45

Title of Topic Topic 1 - 1.46

AdminUser extensions - 6

- Like so:

```
public override Boolean checkPassword(String pass)
{
    return base.checkPassword (pass);
}
```




46

Title of Topic Topic 1 - 1.47

AdminUser extensions - 7

- Note here that we also define this as an **override** function – that’s because C# won’t let us accidentally override a function.
- We have to explicitly indicate our intention.
- Also, to do this, we need to indicate that the method we’re overriding can be overridden.




47

Title of Topic Topic 1 - 1.48

AdminUser extensions - 8

- We do this by giving it the **virtual** modifier.
- We do that in the parent class:

```
public virtual Boolean checkPassword(string pass)
{
    if (pass.ToLower().Equals(password.ToLower()))
    {
        return true;
    }
    return false;
}
```




48

Title of Topic: Topic 1 - 1.49

AdminUser extensions - 9

- We use **base** to refer to the parent class, saying here 'run the checkPassword routine in the parent, and return that as the result'.
- However, we can choose to do that whenever we like, meaning that we can have a checkPassword function that works a bit differently:



49

Title of Topic: Topic 1 - 1.50


AdminUser extensions - 10

```
public override Boolean checkPassword(String pass)
{
    Boolean ret;

    if (locked == true)
    {
        return false;
    }

    ret = base.checkPassword (pass);

    if (ret == false)
    {
        failedLogins += 1;
        if (failedLogins == 3)
        {
            locked = true;
        }
        return false;
    }
    else
    {
        failedLogins = 0;
        return true;
    }
}
```




50

Title of Topic: Topic 1 - 1.51

AdminUser extensions - 11

- Here, we still have the parent class responsible for telling whether a password is correct, but we provide some guard conditions around when that is called.
- If the account has been locked, we don't even bother checking to see if the password is correct.




51

Title of Topic: Topic 1 - 1.52

AdminUser extensions - 12

- The really clever thing about how polymorphism works though is that while it will only allow you to checkPassword if it can guarantee every child of the User class has the method, when it comes to execute the function it'll use the most specialised version of that function available in the object.
- When we call checkPassword on a NormalUser, it'll find the one in the base User class. When we call checkPassword on an AdminUser, it'll find the more specialised one defined in AdminUser.




52


Title of Topic: Topic 1 - 1.53

AdminUser extensions - 13

- As such, we now have objects with two different login regimes that work off of a common stem.
- This is incredibly powerful.
- We'll see more examples as time goes by as to what we can do with this.



53



Awarding
Great British
Qualifications

Topic 8 – Object Orientation (2)

Any Questions?

54
