# COMP 3602

## C# Application Development

## Week Five

# Misc Notes

- **XML Comments – does it still make sense to use inline comments?**

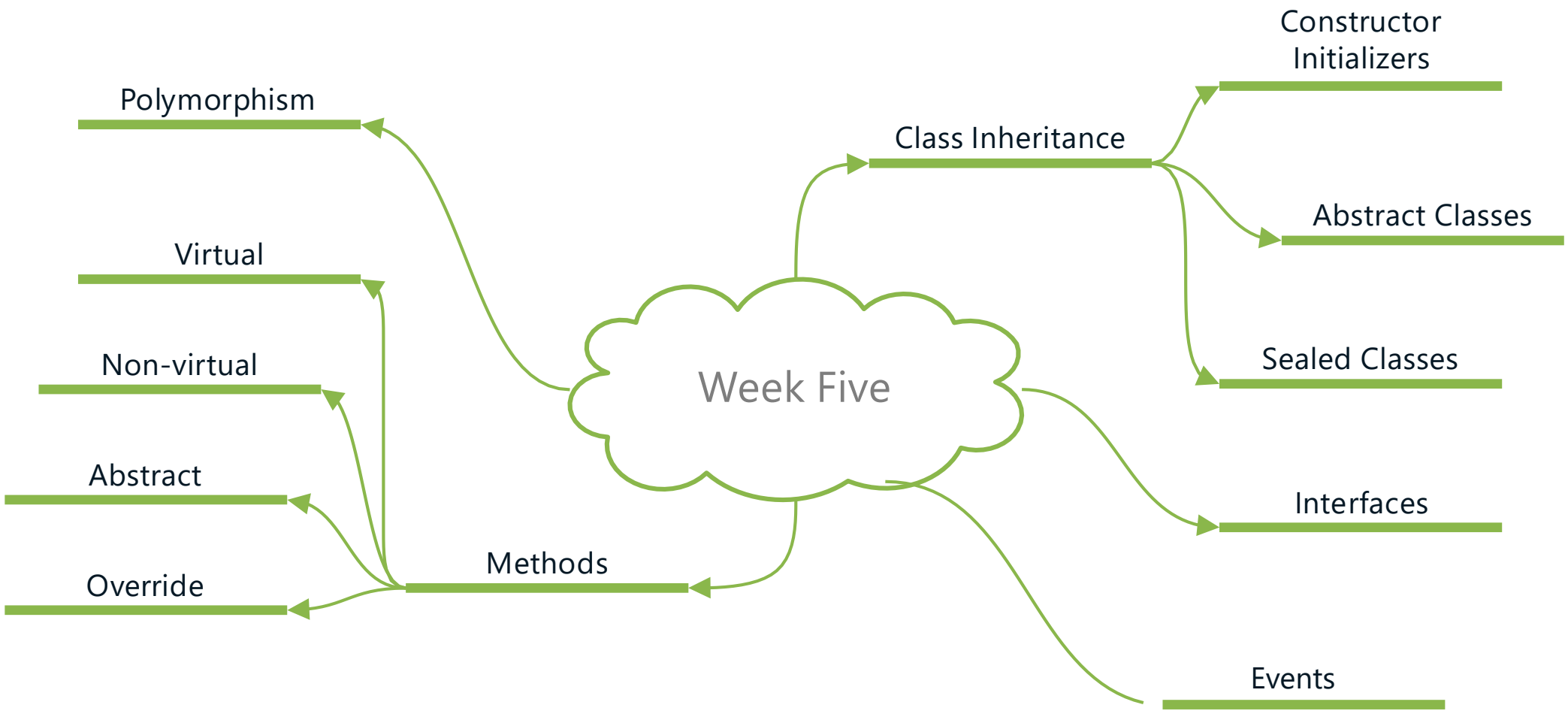
- **Why do we use Property methods (aka properties – get; set;)?**
  - Why not just use methods?

- **Why do we put all code outputting to a console through a dedicated class?**

# Misc. Notes

- *Why is design important?*
  - We want our code to be **readable**
    - *Other people are going to need to work on it*
  - We want our code to be **maintainable**
    - *We are going to need to update it*
  - We want our code to be **flexible**
    - *We are going to need to add to it*
  - We want our code to be **reusable**
    - *Reusable not just within our application, but with others as well (as much as possible/makes sense)*
  - We want our code to be **efficient**
    - *Minimize the work that is done by both us and the code*
    - *Follow best practice first, then performance tune when you have a problem*

# Tonight's Learning Outcomes



# Class Inheritance

- C# like Java, supports single inheritance only.
- C# employs the colon operator in place of Java's extends keyword.

```
class DerivedClass : BaseClass
{
    // implementation
}
```

C#

```
class DerivedClass extends BaseClass
{
    // implementation
}
```

Java

# Class Inheritance

```
2 references
class MyClass
{

}
```

```
2 references
class MyClass : Object
{

}
```

- All C# classes can trace their ancestry back to Object.
- Type Object is the “Mother” of all C# classes.
- When you create a new class, it implicitly derives from Object.

# Class Inheritance

```
2 references
class MyClass
{
    // empty - no implementation
}
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        MyClass myClass = new MyClass();
        myClass.
    }
}
}
```

Equals
GetHashCode
GetType
ToString

bool object.Equals(object obj)
Determines whether the specified object is equ
Note: Tab twice to insert the 'Equals' snippet.

- When a Class is defined, it inherits some functionality from object.
- All methods except GetType() are virtual and can be overridden to alter their default behaviors.

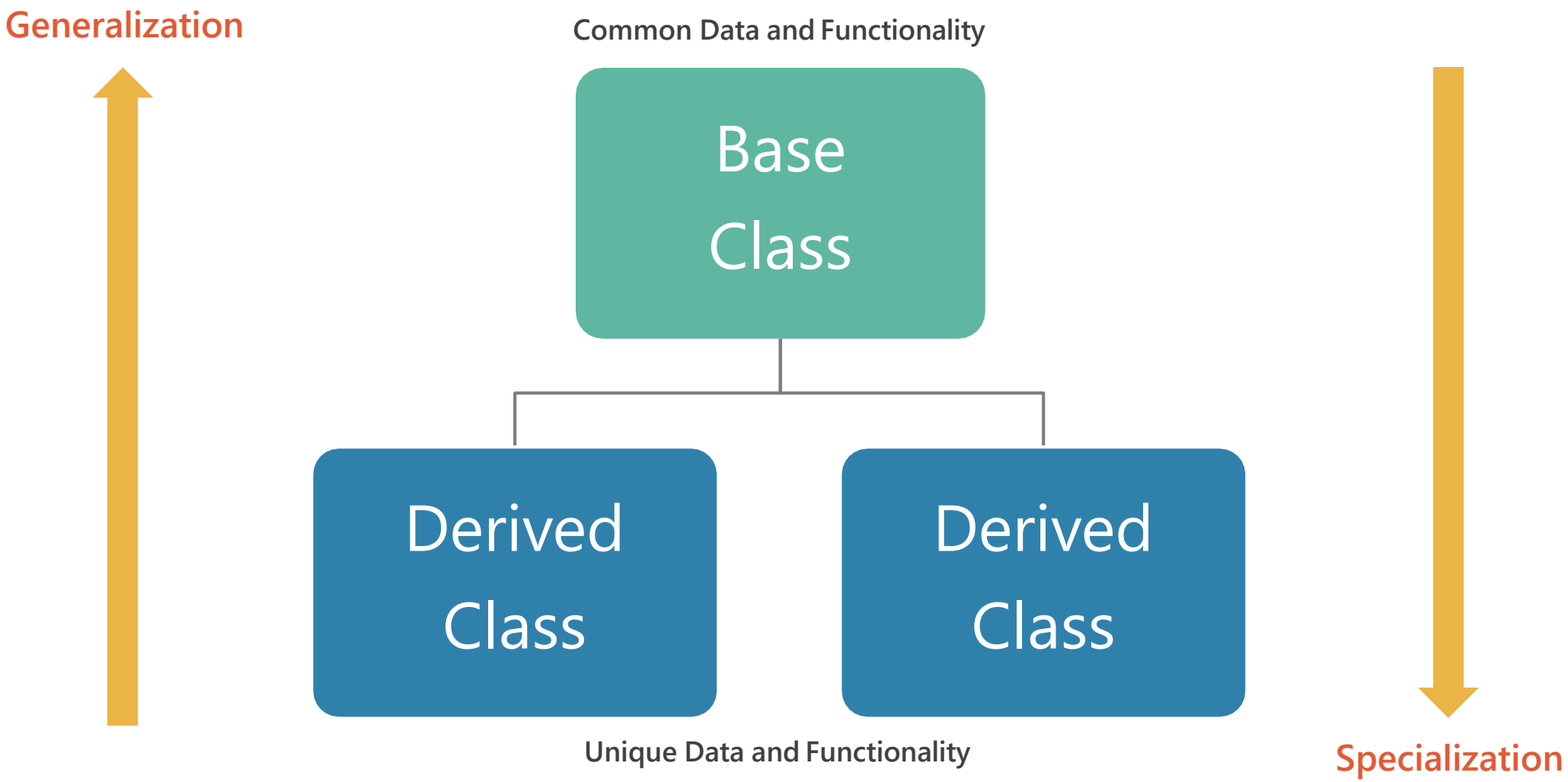
# Constructor Initializers

- Sometimes a derived class' constructor must explicitly invoke one of its base class' constructors or another constructor in the same class.
- In C#, this is done with constructor initializers.

```
2 references
class Item
{
    private int id;

    1 reference
    public Item(int id)
    {
        this.id = id;
    }

    1 reference
    public int Id
    {
        get { return th
    }
}
```

```
4 references
class InventoryItem : Item
{
    private decimal price;
    private string description;

    1 reference
    public InventoryItem(int id, decimal price)
        : base(id)
    {
        this.price = price;
    }

    1 reference
    public InventoryItem(int id, decimal price, string description)
        : this(id, price)
    {
        this.description = description;
    }
```

# Generalization and Specialization



# Virtual and Override Methods

```
1 reference
class BaseClass
{
    0 references
    public void MethodA(string input)
    {
        // implementation
    }

    1 reference
    public virtual void MethodB(string input)
    {
        // implementation
    }
}
0 references
class DerivedClass : BaseClass
{
    0 references
    public override void MethodA(string input)
    {
        // implementation
    }

    1 reference
    public override void MethodB(string input)
    {
        // overidden implementation
    }
}
```

CS0506 'DerivedClass.MethodA(string)': cannot override inherited member 'BaseClass.MethodA(string)' because it is not marked virtual, abstract, or override

- A virtual method in a base class can be optionally overridden in a derived class to extend functionality
- In Java, methods are implicitly virtual and can be overridden in a derived class
- In C#, **a method must be explicitly declared virtual** in the base class to be overridden in the derived class (improves performance of non-virtual methods)

# Abstract Classes and Methods

## Abstract Classes

- Can not be instantiated
- Can contain zero or more abstract methods
- Can contain standard methods
- Can contain virtual methods

```
1 reference
22  abstract class BaseClass
23  {
        0 references
24      public abstract decimal CalculatePay();
25  }
26
    0 references
27  class DerivedClass : BaseClass
28  {
29
30  }
31
```

CS0534 'DerivedClass' does not implement inherited abstract member 'BaseClass.CalculatePay()'

## Abstract Methods

- Contain no implementation
- Can only be placed in an abstract class
- Are implicitly virtual
- MUST be overridden in any derived class
- Cannot have a private scope (public or protected)

# Virtual / Override / Abstract Methods

| Base Class | Derived Class | |
|---|---|---|
| Method Type | Inherit | Override |
| Standard (Non-virtual) | Yes | No |
| Virtual | Yes | Optional |
| Abstract | N/A | Mandatory |

| Protection | Scope / Visibility |
|---|---|
| private | Current class type |
| protected | Current class and derived classes |
| public | Application wide |

# Sealed Classes

```
1 reference
22  sealed class BaseClass
23  {
24      // implementation
25  }
26
0 references
27  class DerivedClass : BaseClass
28  {
29      // implementation
30  }
31
```

CS0509 'DerivedClass': cannot derive from sealed type 'BaseClass'

- Prevents further derivation of the class.
- Equivalent to Java's final keyword.
- Sealed classes in the .NET Base Class Library (BCL) include:

  `System.Console`
  `System.String`

- When a BCL class is sealed, it will indicate so in the MSDN Online Help.

# Interfaces

```
2 references
public interface IDrawable
{
    int penThickness;
    0 references
    void Draw();
}
```

CS0525 Interfaces cannot contain fields

- An Interface defines one or more method signatures with no implementation.
- Any class that implements an Interface MUST provide implementations for each method definition. (a contract)
- Method signatures defined by an Interface are implicitly public.
- Interfaces cannot contain fields.
- Naming convention is to preface the name with a capital “I”.

# Interfaces

```
0 references
public void DrawObject(IDrawable drawableObject)
{
    // implementation
```

Will accept any class that implements IDrawable

```
0 references
public IDrawable CreateDrawable()
{
    // implementation
```

Will return any class that implements IDrawable

- Interfaces can be used as parameter and return types in method signatures.
- Any class implementing that Interface can be passed or returned.

# Interfaces

Classes can extend a single base class and/or implement one or more Interfaces

```
0 references
class DerivedClass : IDrawable
{
    // implementation
}
```

Single Interface

```
0 references
class DerivedClass : BaseClass, IDrawable
{
    // implementation
}
```

Classname must come before interface name

BaseClass + Interface

```
0 references
class DerivedClass : IDrawable, IPrintable
{
    // implementation
}
```

Multiple Interfaces

# Interfaces vs inheritance

***Interfaces*** are used to ensure guaranteed behaviour across *many types* of objects.

E.g. IDrawable, IComparable, IDisposable

Shape, Table, Car, Latte

Shape.Draw()
Table.Draw()
Car.Draw()

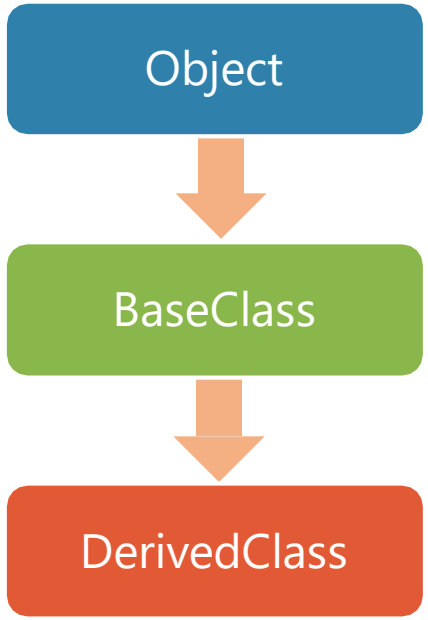
"**Has** a"

***Inheritance*** is used to ensure similar properties and behaviours, while allowing further specialization.

E.g. Pet → Dog → Terrier → Brown Terrier

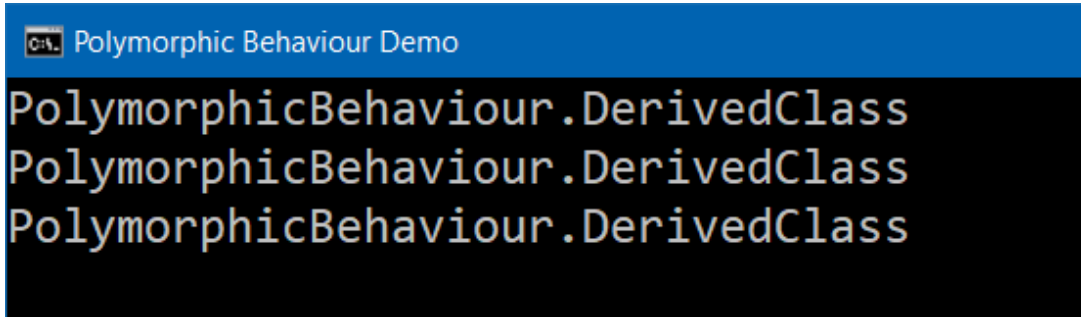
"**Is** a"

# Polymorphism




```
58
59      DerivedClass derivedClass = new DerivedClass();
60      BaseClass baseClass = derivedClass;
61      object objectClass = baseClass;
62
63      Console.WriteLine(derivedClass);
64      Console.WriteLine(baseClass);
65      Console.WriteLine(objectClass);
66
```

The most derived version of a method is always invoked




- The most derived version of a method is always invoked.
- Console.WriteLine() implicitly invokes a class's ToString() method.
- The DerivedClass version of ToString() gets invoked regardless of what type it is cast to.
- Upcasting is implicit (no cast required)

# Events – Overview

**Event Declared**

Public event EventHandler<EventArgs> AccountOverDrawn;

**Event Subscribed**

E.g. account. AccountOverDrawn += AccountWasOverDrawn

**Event Fired**

E.g. AccountOverDrawn.Invoke()

**Event Handled**

E.g. AccountWasOverDrawn(Object sender, EventArgs e) {…}

# Events – Publisher

ChequingAccount Class

-string AccountNumber
-decimal Balance

-void Withdraw()
-void Deposit()

I want to tell other code as soon as an account gets overdrawn

# Events – Publisher - Declare

ChequingAccount Class

-string AccountNumber
-decimal Balance
-**event AccountOverDrawn**

-void Withdraw()
-void Deposit()

So, I'll declare an event!

# Events – Publisher - Fire

ChequingAccount Class

-string AccountNumber
-decimal Balance
-**event AccountOverDrawn**

-void Withdraw()
{
**AccountOverDrawn.Invoke();**
}
-void Deposit()

Then, in some places in my methods, I can **Invoke** the event (aka **fire** the event)

# Events – Publisher

```
class ChequingAccount
{
    1 reference
    public string AccountNumber { get; }
    5 references
    public decimal Balance { get; private set; }


    public event EventHandler<EventArgs> AccountOverDrawn;
```

Publisher

Declare

```
2 references
public void WithDraw(decimal amount)
{
    if (amount <= 0m)
    {
        throw new ArgumentOutOfRangeException("Amount", "Amount must be greater than zero");
    }

    Balance -= amount;

    if (Balance < 0.00m)
    {
        if (AccountOverDrawn != null)
        {
            AccountOverDrawn.Invoke(this, new EventArgs());
        }
    }
}
```

Publisher

Invoke (Fire)

# Events - Subscriber

```
0 references
static void Main(string[] args)
{
    Console.Title = "Banking App - Events Demo";

    while (true)
    {
        ChequingAccount account = AccountUI.GetAccountFromUser();

        if (account == null)
        {
            return;
        }
        else
        {
            account.AccountOverDrawn += AccountUI.AccountWasOverDrawn;
            AccountUI.PerformAccountActions(account);
        }
    }
}
```

Subscriber

Subscribe

```
1 reference
public static void AccountWasOverDrawn(object sender, EventArgs e)
{
    ChequingAccount account = (ChequingAccount)sender;
    Console.WriteLine("  ___   ___  _ __  ___ \r\n / _ \\                                  (_) | (_) | |_) \\__ \\\r\n
    Console.WriteLine($"Account {account.AccountNumber} is overdrawn! Current balance is {account.Balance}.")
}
```

Subscriber

Handle

# Events – Subscriber

BankingApp

-ChequingAccount chequingAccount
- …. otherFields

I know that I am creating an instance of ChequingAccount, and I know it can tell me if it is overdrawn

# Events – Subscriber

BankingApp

-…other methods
- …. other fields

-**void AccountWasOverdrawn()**

So I'll create a method that does some action when the Account is overdrawn

# Events – Subscriber

BankingApp

```
-void Main()
{
  account.AccountOverDrawn += AccountUI.AccountOverDrawn;
}
```

Then, I'll subscribe to the event

# Events – Wired Up

ChequingAccount Class

-string AccountNumber
-decimal Balance
-**event AccountOverDrawn**

-void Withdraw()
{
**AccountOverDrawn.Invoke();**
}
-void Deposit()

BankingApp

-…other methods
- …. other fields

**--void AccountWasOverDrawn()**

# Events - Subscriber

```
0 references
static void Main(string[] args)
{
    Console.Title = "Banking App - Events Demo";

    while (true)
    {
        ChequingAccount account = AccountUI.GetAccountFromUser();

        if (account == null)
        {
            return;
        }
        else
        {
            account.AccountOverDrawn += AccountUI.AccountWasOverDrawn;
            AccountUI.PerformAccountActions(account);
        }
    }
}
```

Subscriber

Subscribe

```
1 reference
public static void AccountWasOverDrawn(object sender, EventArgs e)
{
    ChequingAccount account = (ChequingAccount)sender;
    Console.WriteLine("  ___   ___  _ __  ___ \r\n / _ \\                                  (_) | (_) | |_) \\__ \\\r\n
    Console.WriteLine($"Account {account.AccountNumber} is overdrawn! Current balance is {account.Balance}.")
}
```

Subscriber

Handle

# Events – Publisher

```
class ChequingAccount
{
    1 reference
    public string AccountNumber { get; }
    5 references
    public decimal Balance { get; private set; }


    public event EventHandler<EventArgs> AccountOverDrawn;
```

Publisher

Declare

```
2 references
public void WithDraw(decimal amount)
{
    if (amount <= 0m)
    {
        throw new ArgumentOutOfRangeException("Amount", "Amount must be greater than zero");
    }

    Balance -= amount;

    if (Balance < 0.00m)
    {
        if (AccountOverDrawn != null)
        {
            AccountOverDrawn.Invoke(this, new EventArgs());
        }
    }
}
```

Publisher

Invoke (Fire)