

Clean Classes and Clean Methods



Xavier Morera

Helping .NET developers create amazing applications

@xmorera / www.xaviermorera.com / www.bigdatainc.org



Recommendations and Guidelines



Naming conventions for classes and methods

Best practices for defining classes and methods

Access modifiers

Namespaces

Favoring Readability



Current Situation

Hard to read

Not easy to test

Potential bugs



Favor Readability



Favor Readability



Code should be understood by humans

Use comments as required

- Do not overuse comments

Code lines should be read as an article

- Small explanations when required

**Make it correct, make it clear, make
it concise, make it fast.**

In that order.

Wes Dyer



Favor Readability

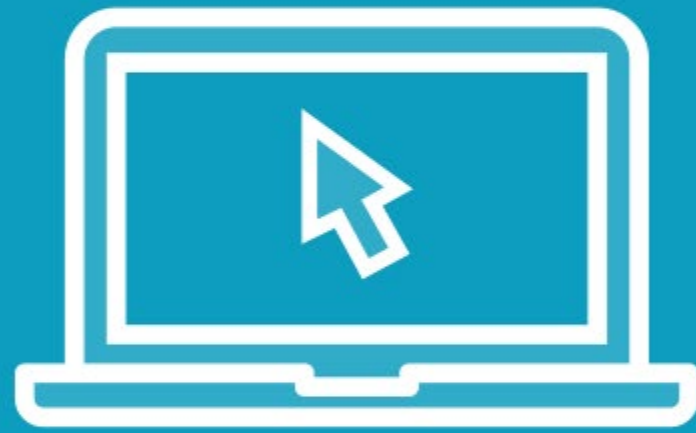
```
public string RateTrail(decimal len, decimal ele, string traffic, string type)
{
    if (len > 0)
    {
        if (ele >= 0)
        {
            if (traffic != null)
            {
                if (type != null)
                {
                    if (len > 20 || ele > 400)
                    {
                        return "hard";
                    }
                    else if ((10 < len && len < 20) || (100 < ele && ele < 400))
                    {
                        if (traffic == "heavy")
                        {
                            return "hard";
                        }
                        else
                        {
                            return "moderate";
                        }
                    }
                }
            }
            else
            {
                return "moderate";
            }
        }
    }
}
```


Favor Readability

It really looks like an arrow!

```
public string RateTrail(decimal len, decimal ele, string traffic, string type)
{
    if (len > 0)
    {
        if (ele >= 0)
        {
            if (traffic != null)
            {
                if (type != null)
                {
                    if (len > 20 || (len < 20 && ele < 400))
                    {
                        return "has a trail";
                    }
                    else if ((10 < len && len < 20) || (100 < ele && ele < 400))
                    {
                        if (traffic.Length > 10)
                        {
                            return "has a trail";
                        }
                    }
                }
                else
                {
                    throw new TrailException("type cannot be null");
                }
            }
            else
            {
                throw new TrailException("traffic cannot be null");
            }
        }
        else
        {
            throw new TrailException("ele cannot be null");
        }
    }
    else
    {
        throw new TrailException("len cannot be null");
    }
}
```

Demo



Favor Readability



Guard Clause



“Fail fast” method

One condition validated at a time

- If fails, execution stops immediately
- Meaningful error is thrown

Easy to read, understand, and maintain

Guard Clause

```
if (length < 0) throw new Exception("Length cannot be null");  
if (length > 255) throw new Exception("Unexpected size");
```



Defining a Class



Defining a Class



What Is a Class in C#?

Data structure that may contain

Data members like constants and fields

Function members like methods, properties, and events

Nested types



Vehicle

Vehicle.cs

```
public class Vehicle
{
    // ...

    public void Move(Key Key)
    {
        // Turn on
        // Move vehicle
    }
}
```


Instantiate
vehicle

```
// ...  
Vehicle vehicle = new();  
vehicle.Move();  
// ...
```

Parts of a Class

Signature

Fields

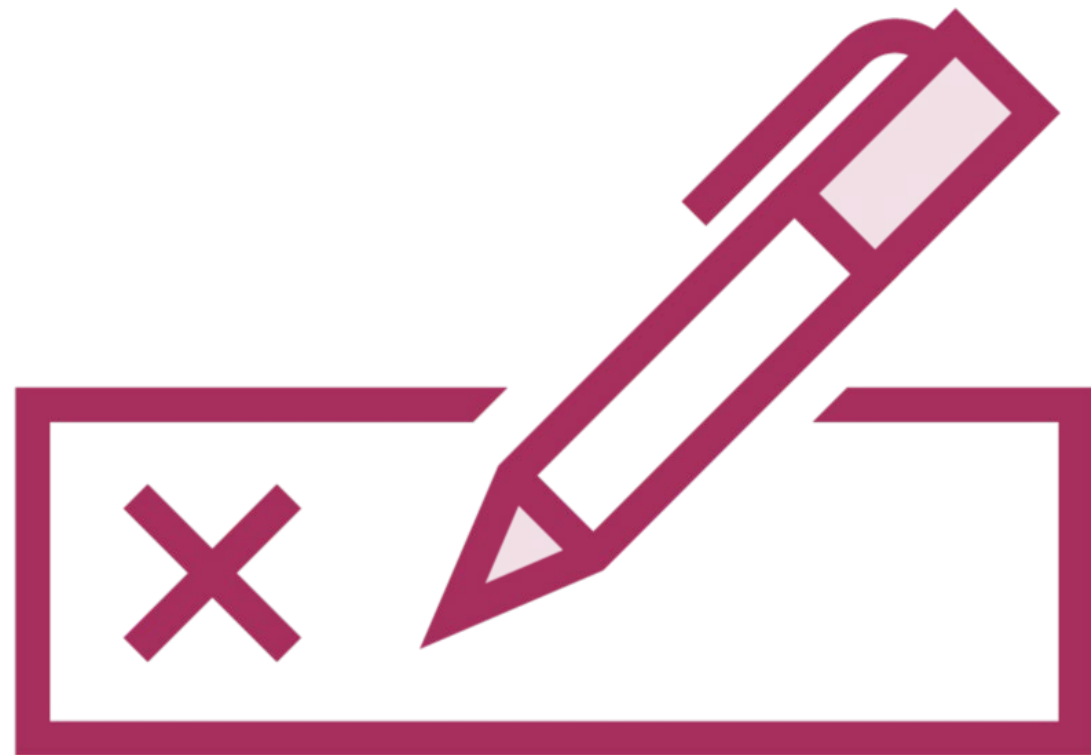
Properties

Constructors

Methods



Signature



Unique identifier for each class

Access modifier

- Inner, public, private...

Class keyword

Class name

Fields and Properties

Field data s are variables that hold

```
private int productId;
```

Property set implements get and

```
public int ProductId  
{  
    get { return productId; }  
    set { productId = value; }  
}
```

Fields and Properties

Fields are variables that hold data

```
private int productId;
```

Property implements get and set

```
public int ProductId  
{  
    get { }  
    set { }  
}
```



Best practice

Use properties to encapsulate fields



Constructors

A constructor is a method that is executed when a new class instance is created

Zero, one, or more constructors

Can have parameters or be parameterless

Typically used to initialize instance fields and properties



Best Practices for Constructors



Default constructor



Provide parameterized constructors



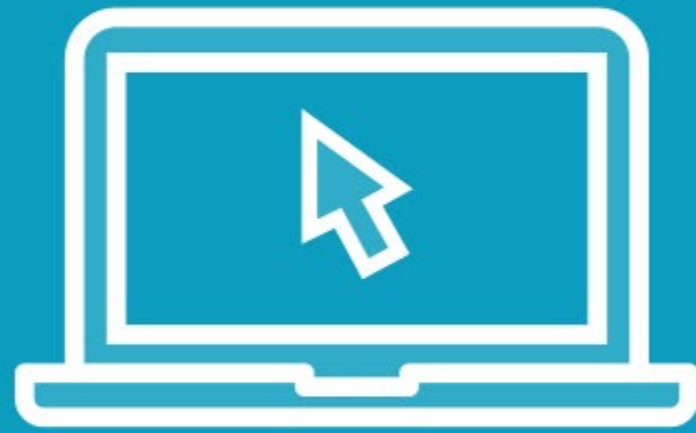
Name the constructors parameters with the same names



Avoid performing heavy work



Demo



Defining Class



Class Naming, Ordering, and Comments



Major Guidelines



Use PascalCasing for naming

Use nouns

Be specific

Single responsibility

Avoid abbreviations

Ordering

- Create one class per code file and avoid large classes

Use XML comments



UniversityStudent

PascalCasing



Student

PascalCasing



Use Nouns



Person

Vehicle

Product

Figure



Be Specific

UniversityStudent

Truck

Kayak

Circle



Use Specific Names

Do 

User
Logging
Email
AmazonBucket
Maps
Report
Forecast

Don't 

WebsiteMu
Utility
Common
XavierFunctions



Single Responsibility

The class must have one and only one purpose



Single Responsibility

The class must have one and only one purpose

```
public class Person
{
    public string FirstName;
    public string LastName;
    public string Id;
}
```

Avoid

**WRONG
WAY**

Abbreviations

Generic suffixes and prefixes

Suffixes and underscores



Ordering

1ST

Fields

2ND

Properties

3RD

Methods



Create one class per code file



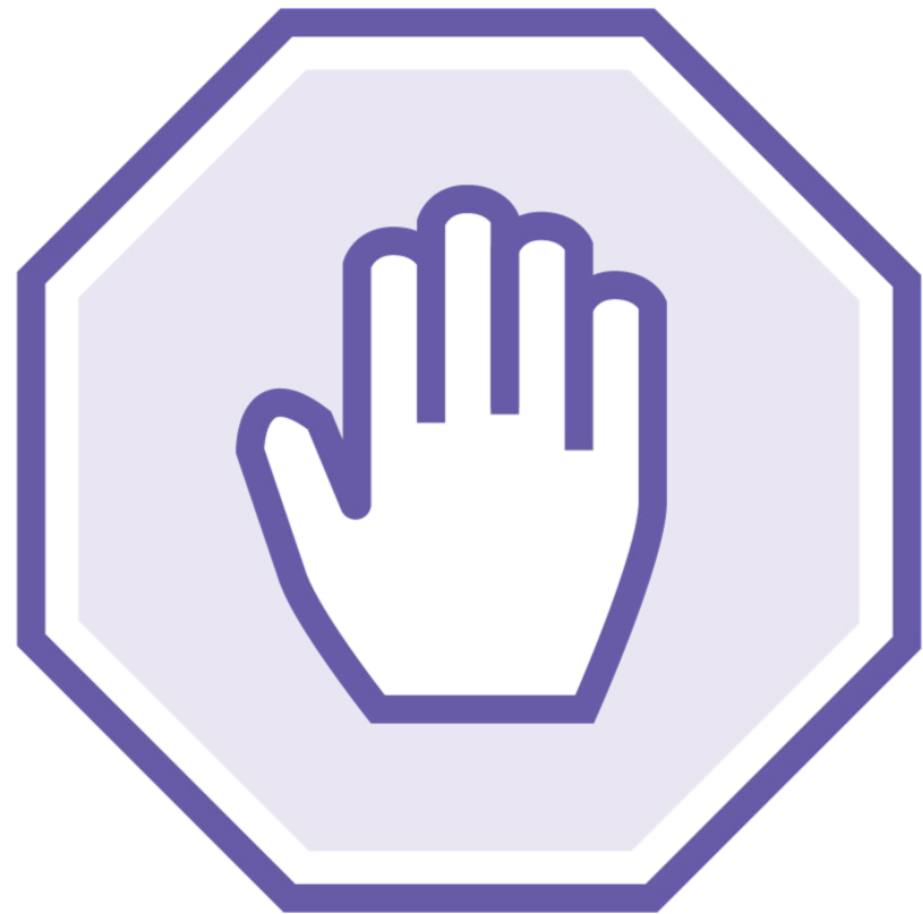
XML document comments are used to define the structure of the output documentation.



XML Comments

Great for documentation

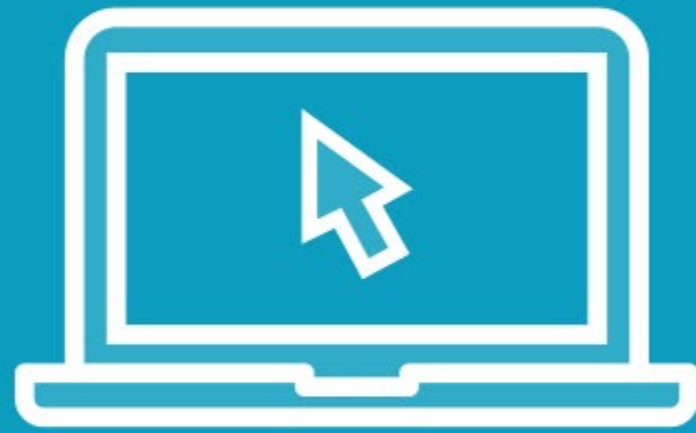
```
/// <summary>
/// Contains the information about a Kayak sold at CarvedRock.
/// </summary>
public class Kayak
{
    // ...
}
```



Avoid large classes



Demo



Class naming and ordering



Static Classes



Static Classes



Great scenarios



Dangerous scenarios



A static class is a special type of class
that cannot be instantiated



Do

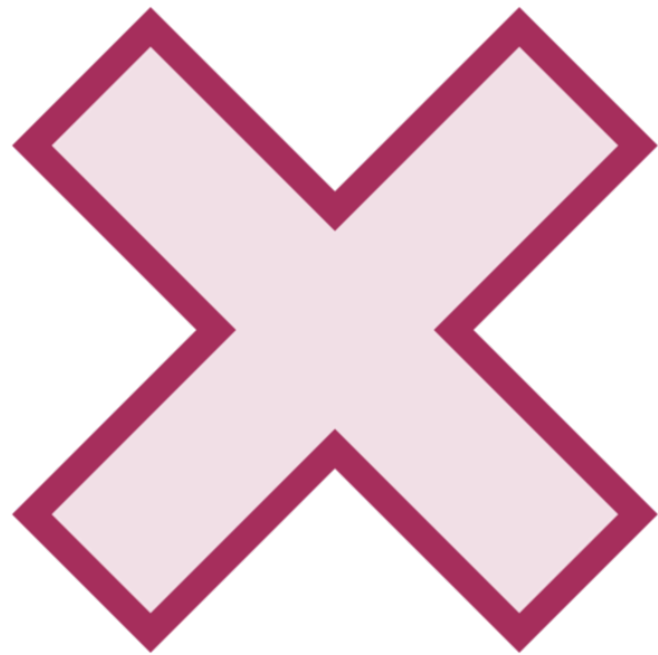


Static classes should be very small and straight to the point

Useful for common code library components



Don't



Use them as function containers

Store state that can be overwritten between calls to the static methods



A Class

```
public class CheckoutFunctions
{
    public decimal CalculateTax(decimal total, string twoLetterStateCode)
    {
        var rate = twoLetterStateCode switch
        {
            // Oregon, Alaska, Montana
            "OR" or "AK" or "MT" => 0.0M,
            // North Dakota, Wisconsin, Maine, Virginia
            "ND" or "WI" or "ME" or "VA" => 00.05M,
            // California
            "CA" => 0.0825M,
            // most US states
            _ => 0.06M,
        };
        return total * rate;
    }
}
```



A Static Class

```
public static class CheckoutFunctions
{
    public decimal CalculateTax(decimal total, string twoLetterStateCode)
    {
        var rate = twoLetterStateCode switch
        {
            // Oregon, Alaska, Montana
            "OR" or "AK" or "MT" => 0.0M,
            // North Dakota, Wisconsin, Maine, Virginia
            "ND" or "WI" or "ME" or "VA" => 00.05M,
            // California
            "CA" => 0.0825M,
            // most US states
            _ => 0.06M,
        };
        return total * rate;
    }
}
```



Static Method

```
public static class CheckoutFunctions
{
    public static decimal CalculateTax(decimal total, string twoLetterStateCode)
    {
        var rate = twoLetterStateCode switch
        {
            // Oregon, Alaska, Montana
            "OR" or "AK" or "MT" => 0.0M,
            // North Dakota, Wisconsin, Maine, Virginia
            "ND" or "WI" or "ME" or "VA" => 00.05M,
            // California
            "CA" => 0.0825M,
            // most US states
            _ => 0.06M,
        };
        return total * rate;
    }
}
```



Invoke Static Method

```
var tax = CheckoutFunctions.CalculateTax(50.0M, "US");
```



Methods and Functions



Methods and functions
are the same in C#



Defining a Method



Code block with a series of statements

- Performs a specific functionality

Called by its name

Declared in a class, struct, or interface

- Access level specified
 - Public, private, protected...
- Optional modifiers like abstract or sealed

Parameters can be by value or reference

Return value specified, including void



Methods

```
/// <summary>
/// Calculates the area of a circle
/// </summary>
/// <param name="radius">Circle radius</param>
/// <returns>The area of a circle with the provided area</returns>
public double CalculateCircleArea(double radius)
{
    const float pi = 3.14F;
    double area = pi * radius * radius;
    return area;
}
```



Clearer Code Intent



Organize code into small and specific pieces

Create well-named methods that represent what each method does

- Use the naming guidelines



Clearer Code Intent

Dirty

```
// If we have a shopping cart with a list of
// products and need to calculate the total to charge:
// <summary>
// Calculate the total of the shopcart
// </summary>
private int total_pricesOfshopcart_list(List<int>
l)
{
    int r = 0;
    for (var i=0;i<l.Count;i++)
    {
        temp = l[i].price;
        r = r + temp;
    }
    return r
}
```

Clean

```
// Clean
public int CalculateTotalShopcart(List<int>
products)
{
    return products
        .Sum(product => product.price);
}
```


Less Code

Dry: Don't repeat yourself



Less code

Don't

```
string season;
// Check if is a winter product:
if (product.winter != "null") {
    season = "winter";
}
// Check if is a spring product:
if (product.spring != "null") {
    season = "spring";
}
// And more repeated code...
// Check if it's a valid winter category
product
if (season != "winter"){
    return false;
}
```

Do

```
private bool ValidProductRequest(Product
product, string seasonCategory)
{
    // Additional checks can be included
    if (product.seasonCategory ==
seasonCategory)
    {
        return true;
    }
}
```

Better indentation



Good Methods (Usually) Fit in a Screen

Original.cs

```
if (AddProduct(productId))
{
    if (changeInventory)
    {
        // Many statements
        // Many statements
        // Many statements
        // Many statements
    }

    if (notifyCompany)
    {
        // Many statements
        // Many statements
    }
    // Many statements
}
```

Nice.cs

```
if (AddProduct(productId))
{
    if (changeInventory)
        ChangeProductInventory(productId);

    if (notifyCompany)
        SendNotification(productId);

    ModifyBill(addToBill, productId);
}
```

Flag Arguments

Don't

Flag arguments may be a bad sign

```
// Don't
private void AddProduct(Id ProductId, bool
changeInventory, bool notifyCompany, Bill
addToBill)
{
    // Possibly, sending emails, changing
store inventory and billing are separate
concerns from adding a product.
}
```

Use as few parameters as possible

```
// Do
private void AddProduct(Id ProductId)
{
    // add product
}
private void ChangeInventory(Id ProductId)
{
    // etc... and maybe the best place for
some of these is another class
}
```

Flag Arguments

Do

```
if AddProduct(productId)
{
    if (changeInventory)
        ChangeProductInventory(productId);

    if (notifyCompany)
        SendNotification(productId);

    ModifyBilll(addToBill, productId);
}
```



Fail Quickly

Do guard clauses, again

Don't

```
// Don't
private void RegisterRoute(string location, string
accessibility)
{
    if (!string.IsNullOrEmpty(location)) {
        if (!string.IsNullOrEmpty(accessibility)) {
            // register the new route
        } else {
            throw new ArgumentException("Location is
required.");
        }
        throw new ArgumentException("Accessibility
details are required.");
    }
}
```

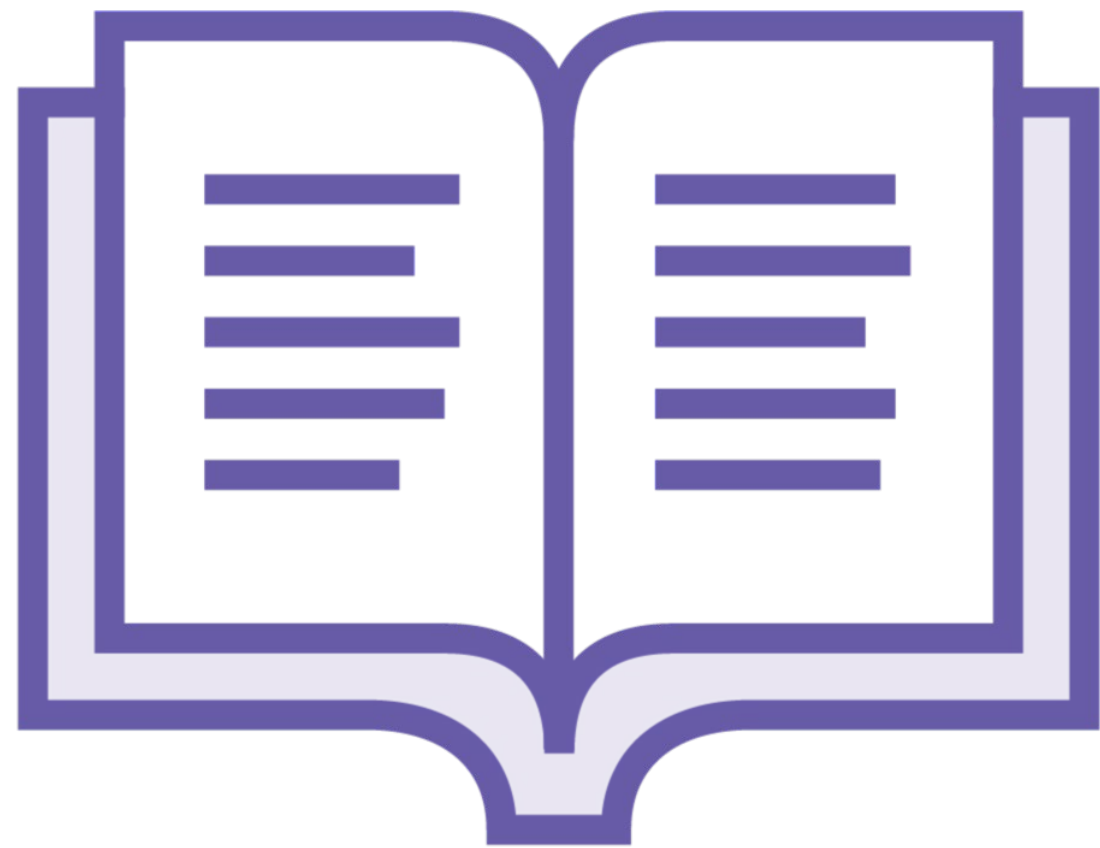
Do

```
// Do
private void RegisterRoute(string location, string
accessibility)
{
    if (string.IsNullOrEmpty(location))
    {
        throw new ArgumentException("Location
required.");
    }
    if (string.IsNullOrEmpty(accessibility))
    {
        throw new ArgumentException("Accessibility
details required.");
    }
    // register the new route
}
```

Major Method Naming Guidelines



Naming Guidelines



Use verbs that indicate action

- Search, Find, Register, Login, Validate, Save...

But avoid generic ones like do or use

Use get only when method has constant time complexity

- Retrieves a value without performing calculations

Boolean methods should begin with is, are, was, were...

Naming Guidelines



Do not describe arguments

- findUserByUserIdAndToken
- Can be simplified to
 - findUser(User userId, Token token)

Avoid abbreviations

- GetIdById
- Better GetPatientIdByCaseId

Methods use PascalCasing

Naming Guidelines

Always add XML document comments

```
/// <summary>
/// Validate the information about the payment status of the order sold on CarvedRock
/// </summary>
/// <param name="orderId">Purchase order Id</param>
/// <returns>True if Payment Status is valid, False if not</returns>
public bool ValidatePaymentStatus(Id orderId)
{
    // Complicated code here
}
```

Namespaces and References



Namespaces are heavily used in C#
programming in two ways

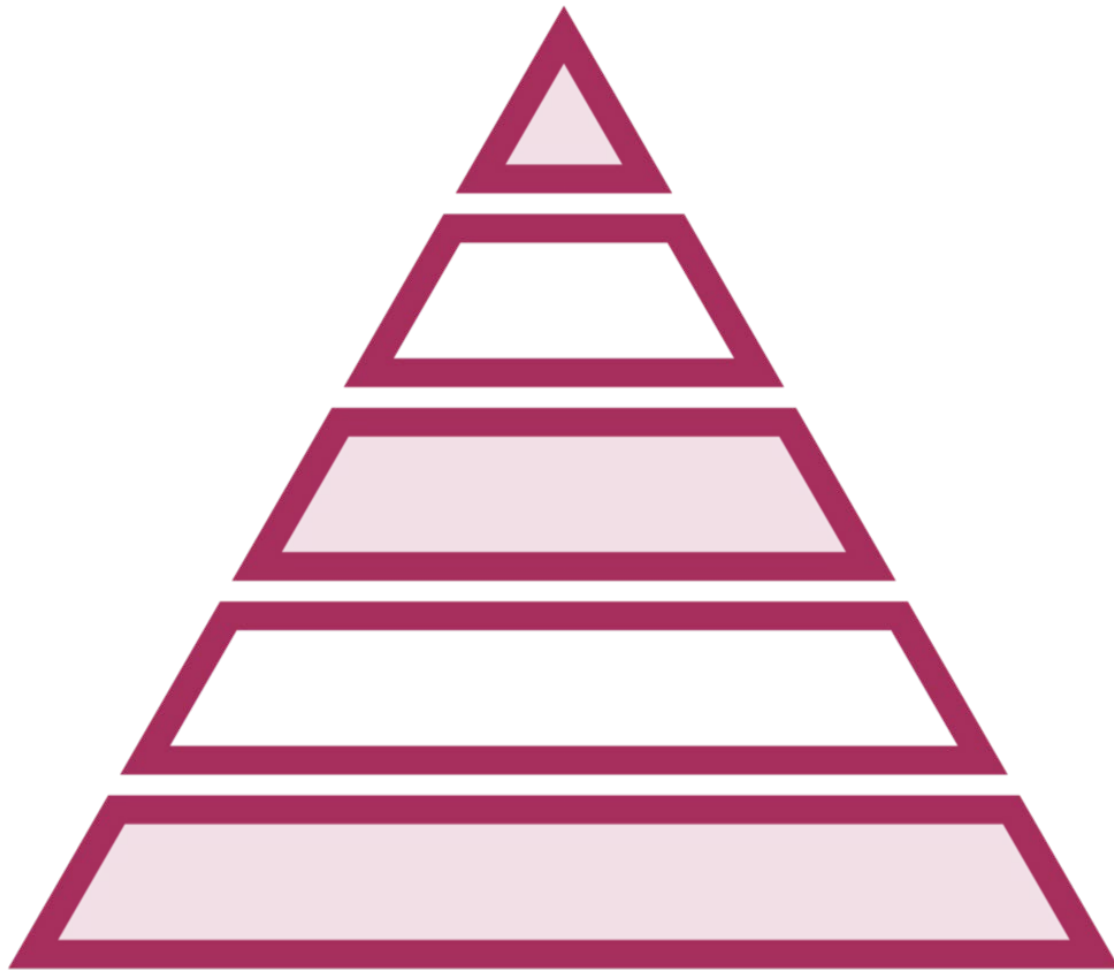




Namespace

Most of the classes under `System.*` and `Microsoft.*` is what's known as the .NET Base Class Libraries





Declaring your own namespaces can help you control the scope of class and method names in larger programming projects



Namespace

Namespaces are declared like this

```
namespace CarvedRock.BL
{
    // class ... { ... }
}
```

Namespace

Namespaces are declared like this

```
namespace CarvedRock.UI.MAUI
{
    // class ... { ... }
}
```

```
namespace CarvedRock.UI.Web
{
    // class ... { ... }
}
```

Namespaces

Do

**Follow
<company>.<technology>.
<feature>**

Use Pascal Casing

Don't

**Use System or Microsoft in
your own classes**

**Use a class name in the
namespace**



References helps you use components
from other namespaces



References

Do

References must be ONE-WAY

Don't

**Avoid excessive use of the
'using static' directive**



Takeaway



Code should be understood by humans

Make it correct, make it clear, make it concise, make it fast

- In that order

Never create “arrow code”

- Use guard clauses instead
- Fail fast



Takeaway



Class has

- Data members
- Function members
- Nested types

Includes

- Signature, fields, properties, constructors, and methods



Takeaway



Naming guidelines

- PascalCasing
- Be specific, single responsibility, avoid abbreviations
- One file per class
- Ordering
 - Fields, properties, and then methods



Takeaway



A method and a function in C#

- Are the same

Help organize functionality

Follow naming guidelines

- Use verbs that indicate action
- Avoid generic verbs
- “Get” only with constant time complexity
- Boolean methods
 - Start with is, are, was, were



Takeaway



Use static classes for common code library components

- Beware of storing state

Use namespaces to organize functionality

- `<company>.<technology>.
<feature>`

