

C# for HAAS F1

The Live Event



Pete Behague

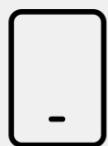
Principle Technical Learning Specialist



Peter.Behague@qa.com

QA

Housekeeping



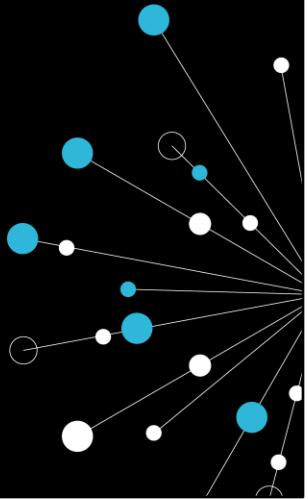
QA

3

Session Content

- Course administration
- Pre-requisites
- Course objectives
- Course outline
- Introductions
- Questions
- Allocation of Virtual Machines

QA





QA

Housekeeping slide

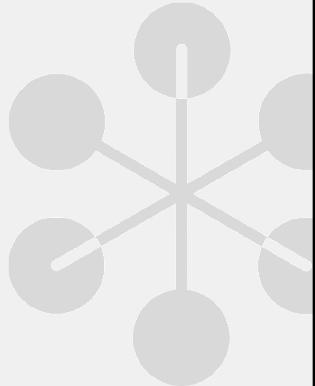
Course Prerequisites

Essential skills:

- Prior experience in programming:
 - Declaring variables
 - Writing loops
 - Passing arguments / parameters
 - Invoking functions

Beneficial skills

- Awareness of object-oriented principles
 - Objects have 'state' and 'behaviour'
- Working with a modern IDE: Visual Studio, Eclipse, etc.



QA

Course Objectives

To enable you to use C# and .NET Core to:

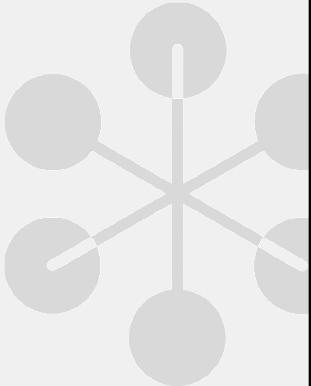
- Gain a solid understanding of object-oriented programming (OOP) to the level of defining classes, methods and properties.
- Be able to create simple unit tests
- Navigate the Visual Studio Integrated Development Environment (IDE) and make use of its code completion features
- Adhere to best practice by using SOLID principles
- Recognise and use a number of commonly used coding patterns
- Understand the use of delegates and lambdas to reduce code duplication
- Be able to create Language Integrated Queries (LINQ)
- Understand how to handle exceptions in C#



QA

Course Outline

- Introduction to Git and GitHub
- Review of OOP Principles and LINQ
- Inheritance and Abstract classes
- Interfaces
- SOLID Principles
- Design Patterns
- Delegates and Lambdas
- Exception Handling
- Unit Testing



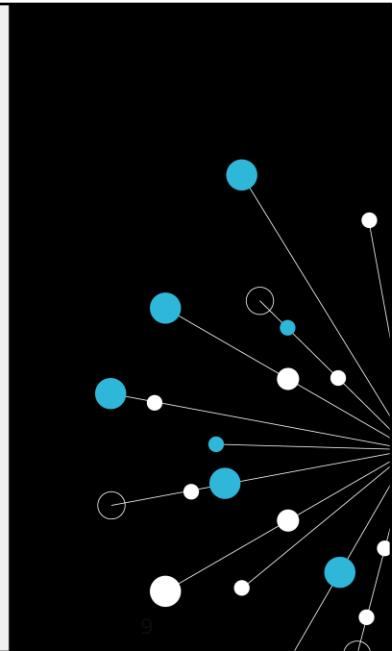
QA

8

Activity: **Introductions**

Preferred name
Organisation & role
Experience of C# and OOP
Key learning objective
Hobby/Outside interest

QA



9

Questions

Golden rule

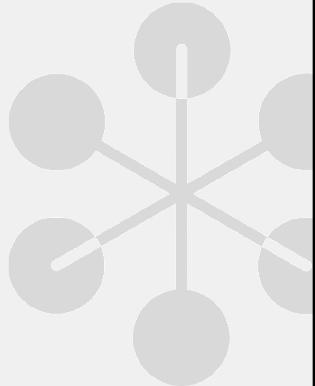
- 'There is no such thing as a stupid question'

First amendment to the golden rule

- '... even when asked by an instructor'
- Please have a go at answering questions

Corollary to the golden rule

- 'A question never resides in a single mind'
- By asking a question, you're helping everybody



QA

10

Allocation of Virtual Machines

A number of labs make use of some hefty pieces of software which eat up processor power and memory.

For this reason, your tutor will allocate you a virtual machine which you are expected to make use of.

The virtual machine has all the necessary software that you need for the course already installed on it.

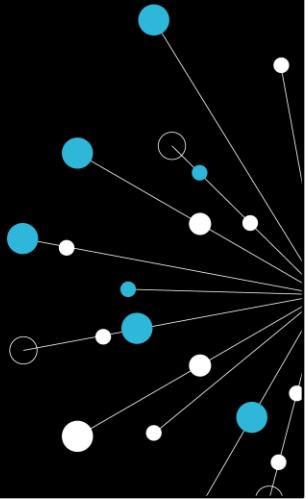


QA

Summary

- Course administration
- Pre-requisites
- Course objectives
- Course outline
- Introductions
- Questions
- Allocation of Virtual Machines

QA



Introduction to Version Control using Git, GitHub and VSC

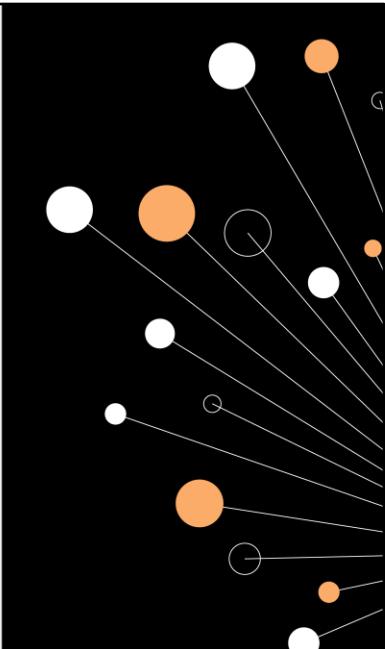
C# for HAAS F1



Learning objectives

To understand the need for source control and how to use Git, GitHub and Visual Studio Code to manage a code repository

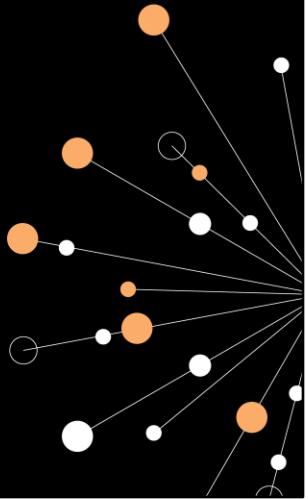
QA



Session Content

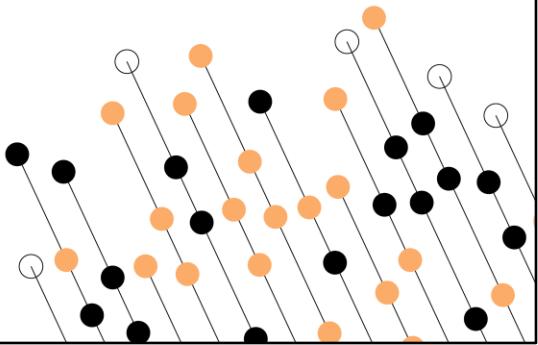
- Introduction to Version Control
- Overview of Git and GitHub
- Installing and Configuring Git
- Basic Git Commands
- Introduction to GitHub
- Using Git with Visual Studio Code

QA



Introduction to Source Control

Introduction to Version Control using Git, GitHub and VSC

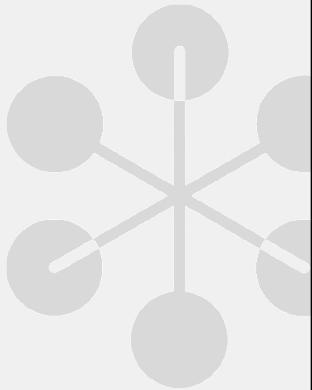


Introduction to Source Control

It is the practice of tracking and managing changes to code.

Known by a few different names:

- version control
- revision control



QA

Vital for managing software development projects

Being able to track changes to code allows developers to:

- Centralise all code changes and additions to one code repository
- Allow for simple and effective collaboration within development teams
- Control the integration of new code into the codebase
- Track changes from the entire team over the full lifetime of the project
- Revert code back to previous versions



QA

Source Control Management (SCM)

Used by developers to implement source-control practices giving the ability to:

- Store code in a central repository
- Track changes over time
- Create code branches so that additions are made in isolation from stable code
- Merge new code into a stable release branch, known as the main branch
- Integrate with CI/CD automation tools (e.g. Jenkins and CircleCI) such that code will be built and tested as it is generated and pushed to the repository

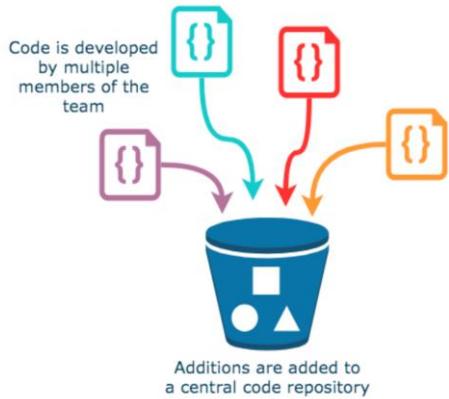


QA

Repositories

Having one place where your code resides means that versioning stays consistent, regardless of who's working on it.

However, there is a danger developers could overwrite another developer's changes. So...

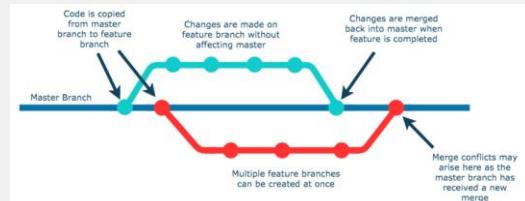


QA

Branching

Allows for each developer to work on new features in isolation by creating a version of the source code that only they are working on

Note: Whilst an important feature further discussion is beyond the scope of this introduction.



QA

Without source control, having multiple developers developing on the same code base simultaneously inevitably results in code conflicts and breakages to functionality, as two developers may need to work on the same parts of code to create their particular features.

Branching allows for each developer to work on new features in isolation by creating a version of the source code that only they are working on.

This new version is called a feature branch, while the source code lives on the main branch.

Once changes have been made and the feature is implemented, the feature branch code is merged back into the source code on the main branch.

The SCM system will automatically detect the differences between the feature and main branches and alert the developer if there are any conflicts.

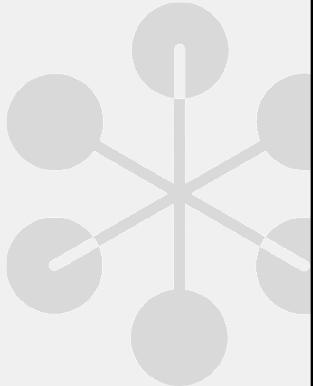
Conflicts may arise if changes have been made to the main branch after the feature branch has been created. The SCM will clearly show the changes that need to be made before merging can occur.

Code Tracking

Code tracking allows development teams to keep track of all changes made to a project over time.

Problems easily solved as code contributors can be consulted directly regarding their code

If a feature causes a bug in the software after it is merged to main, code tracking makes it easy to revert the main branch back to a previous stable state



QA

Code tracking allows development teams to keep track of all changes made to a project over time. This allows for greater organisation, as all additions to code are fully documented and attributed to the developer who made them.

Problems can then be solved with ease as code contributors can be consulted directly regarding their code; if a contributor is no longer a part of the team and can't be consulted directly, their changes are still fully documented allowing the team to track their contributions despite their absence.

If a feature causes a bug in the software after it is merged to main, code tracking makes it easy to revert the main branch to a previous stable state until fixes can be made.

Git Tools and Services

SCM Tools

- Git (Most Common)
- Mercurial
- Subversion (often abbreviated to SVN)
- CVS
- Perforce

Repository Hosting Services

- GitHub
- GitLab
- Bitbucket
- SourceForge
- Launchpad

QA

Git is by far the most common SCM system used in software development. Git is a free and open-source version control system.

Its ubiquity is largely due to how easy it is to learn and its tiny footprint, which is a result of being written in low-level C code.

It is a decentralised SCM tool, meaning its operations are largely performed on your local computer and requires no communication with an external server, resulting in very fast performance.

Many cloud providers have their own code repository offerings which offer simple and powerful integration with other cloud services:

AWS CodeCommit

Azure Repos (as part of Azure DevOps)

Google Cloud Source Repositories

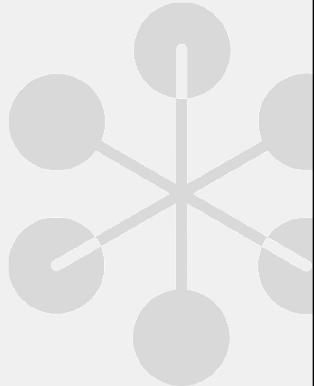
Git

Git is a distributed version control system (VCS) used for tracking changes in files and coordinating work on those files among multiple people.

It was created by Linus Torvalds in 2005, primarily to manage the development of the Linux kernel.

Most widely used version control system in software development.

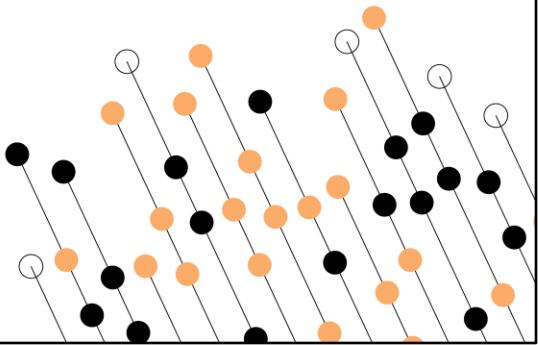
Versions are available for Windows and Mac operating systems.



QA

Overview of Git and GitHub

Introduction to Version Control using Git, GitHub and VSC



GitHub and Git Bash

GitHub

Cloud-based code hosting platform to help developers store, manage, track and control changes to their code.

Used for collaboration and version control.

Git Bash

Command prompt where Git instructions can be used to interact between GitHub and a local copy of the code.

Can use Bash CLI (Command Line Interface) in Windows or Mac.

Git Bash allows you to interact with the Windows environment using Linux commands and comes preloaded with Git for source control.



QA

GitHub is a cloud-based code hosting platform to help developers store, manage, track and control changes to their code. It is a platform used for collaboration and version control.

Git Bash is the command prompt used to interact between GitHub and a local copy of the code. It is a powerful and easy to use Bash CLI (Command Line Interface) in Windows or Mac, however, it is recommended to use the default terminal in Mac.

Git Bash allows you to interact with the Windows environment using Linux commands and comes preloaded with Git for source control.

Getting started with GitHub

Need to have an account:

<https://docs.github.com/en/get-started/start-your-journey/creating-an-account-on-github>

Create a repository and use it to merge changes into the main branch from another:

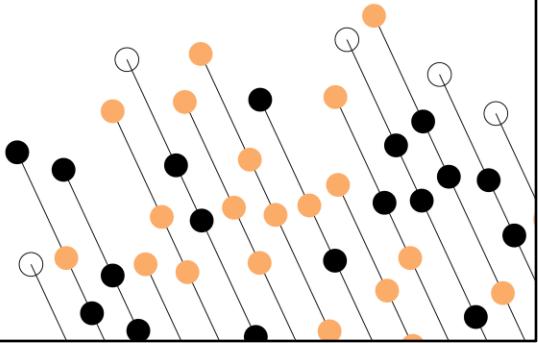
<https://docs.github.com/en/get-started/start-your-journey/hello-world>



QA

Installing and Configuring Git

Introduction to Version Control using Git, GitHub and VSC



Git Bash

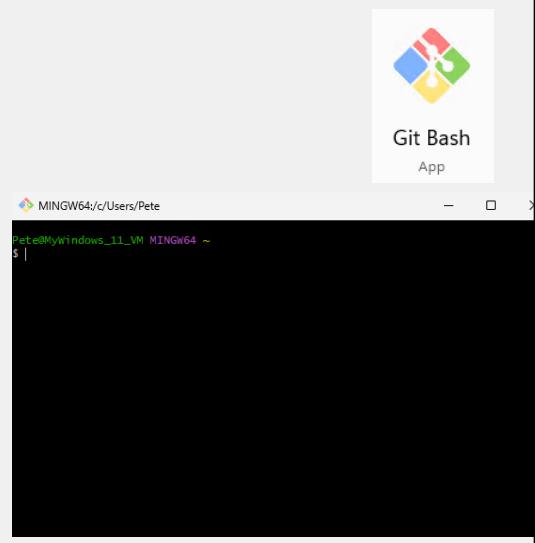
In your browser of choice visit Git Bash download:

<https://git-scm.com/downloads>

Download appropriate version for your operating system

Find the installer that was downloaded and run it.
Then take the default options

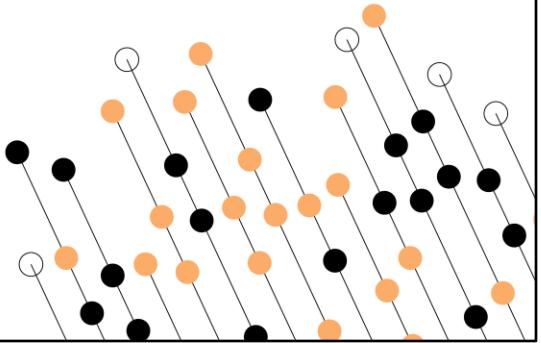
Git Bash app can be launched in the conventional way (Windows Start button...)



QA

Basic Git Commands

Introduction to Version Control using Git, GitHub and VSC



Git Commands – Entomology

Git commands all follow the same convention:

- The word 'git'
- Followed by an optional switch
- Followed by a Git command (mandatory)
- Followed by optional arguments

```
git [switches] <commands> [<args>]
```

```
git -p config -global user.name "Dave"
```



QA

The -p switch paginates the output if needed.

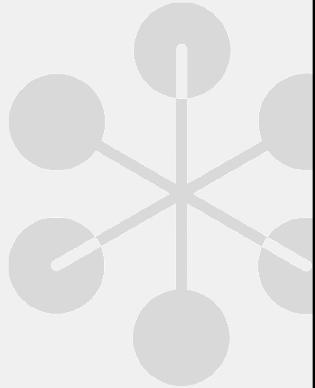
Getting started with Git

When you first launch Git you will be at your home directory

- ~ or \$HOME

Through the Git Bash you can then move through and modify the directory structure

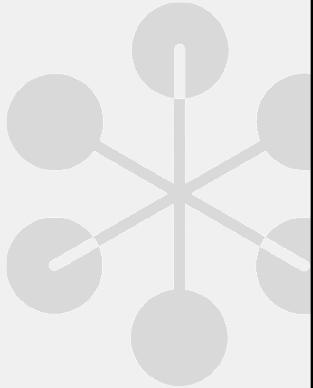
Command	Explanation
ls	Current files in the directory
mkdir	Make a directory at the current location
cd	Change to the current directory
pwd	Print the current directory
rm	Remove a file (optional -r flag to remove a directory)



QA

Git Help

```
git help
```

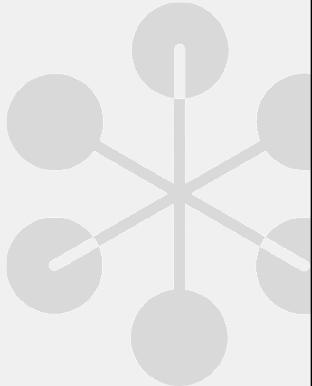


QA

Git Help – Quick Lab

Use your command line to access git help find out about:

- help
- glossary
- -a
- config
- -g



QA

GIT Key Concepts - Repos

GIT holds assets in a repository (repo)

- A repository is a storage area for your files
- This maps to a directory or folder on your file system
 - These can include subdirectories and associated files
 - Relevant Windows command prompt commands include:

```
md firstRepo  
cd firstRepo  
git init
```

The repo requires no server but has created a series of hidden files

- Located in .git folder

```
git status
```



QA

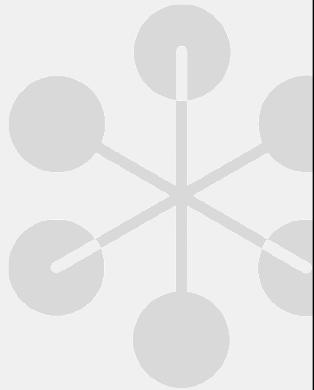
Adding Files - 1

You can see the status of your git repository

```
git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
```



QA

Adding Files - 2

To add a new file, use the 'add' argument

```
# a single file  
git add specific_file_name.ext  
  
# To add all the files  
git add .  
  
# add changes from all tracked and untracked files  
git add --all
```

Git status will show the newly added file

```
On branch master  
No commits yet  
  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  new file:   yellow/New Text Document.txt
```

QA



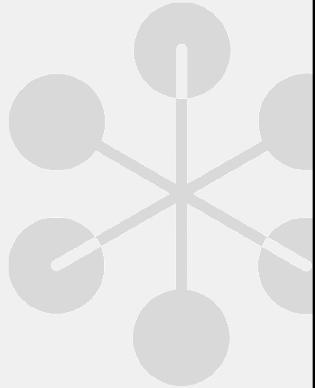
Committing files

To commit the changes, use the “commit” argument

- You can specify the author with the –a flag
- The –m flag is used to set a message

```
$ git commit -m "More content for DG02 - git"  
[master 93e8300] More content for DG02 - git  
1 file changed, 0 insertions(+), 0 deletions(-)
```

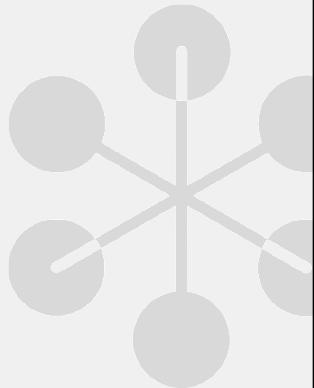
This is now saved to the local version of your repository



QA

Making a directory – Quick Lab

- Launch your git command line (GitBash)
- Ensure you are at your home directory
- Find the corresponding directory using Window's File explorer
- Create a directory called GitTest
- Open a Windows Command prompt and navigate within the directory using the command line
- Run the git init command
- Use Windows File Explorer to create a sub-folder
- Use Windows File Explorer to
- Add a file to the new folder
- From the command prompt type and run git add .
- Enter and run git status and note that your new file is being tracked



QA

Creating a Git Repo – Quick Lab Continued

Enter `git commit`

Enter `git status`

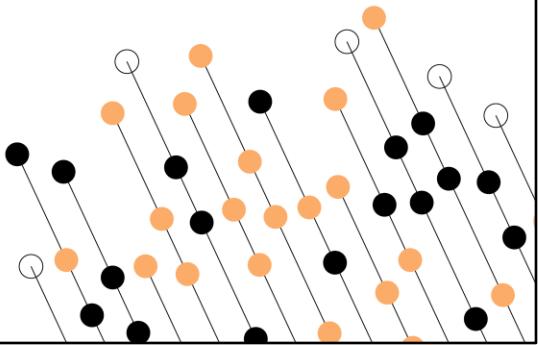
Note the new files has been saved to the local repository



QA

Introduction to GitHub

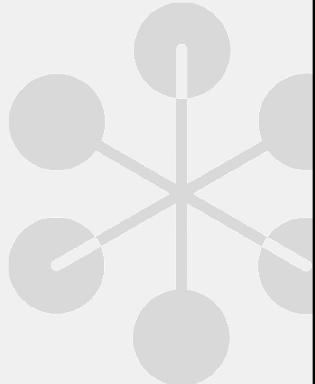
Introduction to Version Control using Git, GitHub and VSC



GitHub

Remember, a GitHub repository will::

- Allow you to create copies of your local repositories to which you can:
 - Invite others to edit and make changes
 - Pull copies of the current state of the repository back down to the local one
 - You will be warned about any conflicts
 - Push local changes up to the remote repository



QA

42

Attack of the Clones!

Cloning makes a physical copy of a Git repository

- It can be done locally or via GitHub
- You can push and pull updates from the repository

The benefit of cloning repos is that the commit history is maintained

- Changes can be sent back between the original and the clone

Cloning is achieved with the clone command:

- Or through the GUI

```
$ git clone source destination_url
```

The GUI branch visualiser gives us a very useful way to see the origins of branches (e.g. run `gitk` command)



QA

Cloning a repository

Cloning copies the entire repository to your hard drive

- The full commit history is maintained

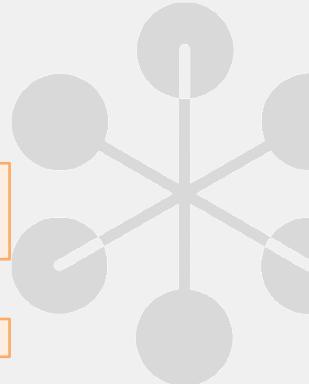
To clone a remote repository:

```
$ git clone [repository]
```

```
$ git clone https://github.com/username/repositoryname
```

To clone a specific branch:

```
$ git clone -b branchName repositoryAddress
```



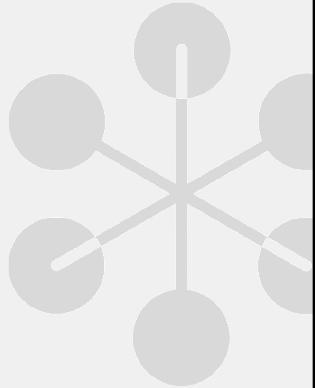
QA

Staging a Change

Staging is the step that you must take before committing a change

Enables you to choose what changes are actually going to get committed to the repository

```
# stage all files  
git add --all  
  
# stage all files (from the root of your project)  
git add .  
  
# stage selected files: git add [FILES]  
git add file_1.txt file_2.txt  
git add *.txt
```



QA

Committing a Change (`git commit`)

When you make a commit to a Git repository, you are effectively saving the changes that you have staged to the repository

Git requires a message to be saved against the commit

- This message is important, so that you and others can understand what it is that you changed

Commits can be reverted

```
# git commit -m "[COMMIT_MESSAGE]"  
git commit -m "initial commit"
```



QA

Working with remote repositories

To see configured remote repositories run the `git remote` command

If you have cloned a repository, you should be shown the origin.

To add a repository:

```
$ git remote add [shortname] [url]
```

Shortname becomes an alias to the repository.

When you have your local copy project at a point you want to share, you have to push it upstream:

```
$ git push origin master
```



QA

You can also specify `-v`, which shows you the URL that Git has stored for the short name to be expanded to.

If you want to rename a reference, in newer versions of Git you can run `git remote rename` to change a remote's shortname.

```
$ git remote rename old_name new_name  
$ git remote  
origin  
new_name
```

It's worth mentioning that this changes your remote branch names, too. What used to be referenced at `old_name/master` is now at `new_name/master`.

If you want to remove a reference for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore — you can use `git remote rm`:

```
$ git remote rm new_name
```

```
$ git remote  
origin
```

Pulling the repository

To retrieve all the changes made to a repository we use the `pull` command

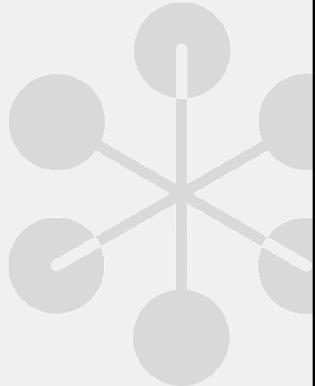
```
$ git pull
```

It is good practice to pull the repository before pushing changes

You get an up-to-date copy of the repo to push to

You can see any conflicts before they are pushed

You can stash your changes before pulling the remote branch



QA

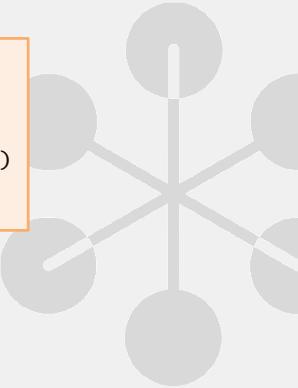
Pushing to the repository

To update the changes in the remote repository, use the push command

```
$ git push origin master  
Enumerating objects: 5, done.  
Counting objects: 100% (5/5), done.  
Writing objects: 100% (3/3), 272 bytes | 136.00 KiB/s, done.  
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)  
To https://github.com/username/demo.git  
    720b1f3..ecf0332  master -> master
```

To set the branch to push automatically use:

```
$ git push --set-upstream origin master
```



QA

Ignoring Files

Git categorises all files in your local repository into:

Tracked - files which have been staged or committed

Untracked - files which have not been staged

Ignored - files which will be ignored

We often do not want some files in our remote repository. e.g. compiled code, build output directories, dependency caches.

In order to make sure that Git does not track them, we need to create a **.gitignore** file in our repository.

There are sets of standard prepopulated .gitignore files for different programming languages.



QA

Git categorises all files in your local repository into "tracked" (files which have been staged or committed), "untracked" (files which have not been staged), or "ignored" (files which will be ignored).

We often do not want some files in our remote repository. This could be because they are large or unnecessary (e.g. compiled code, build output directories, dependency caches). In order to make sure that Git does not track them, we need to create a **.gitignore** file in our repository.

This file is a list of intentionally untracked files that Git should ignore. We can use an asterisk "*" to specify which type of files should be ignored (an asterisk matches anything except a slash).

For example, if we use "`*.pyc`", Git will ignore all files that end in ".pyc".

A leading "`**/`" means match in all directories. For example, if we use "`**/venv`", Git will ignore a file or directory "venv" anywhere in our repository.

Our **.gitignore** file may look like one of these examples:

```
# gitignore file example 1
```

```
**/venv
```

```
*.pyc
```

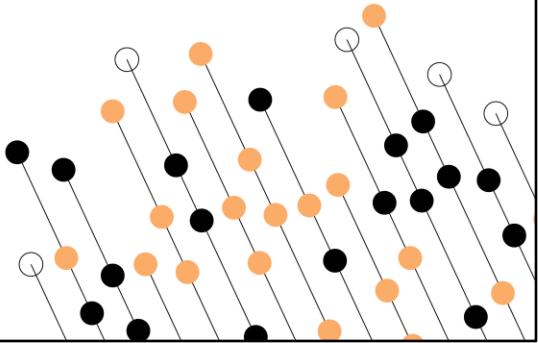
```
__pycache__
```

```
# gitignore file example 2
```

.settings/
target/
bin/

Using Git with Visual Studio

Introduction to Version Control using Git, GitHub and VS



Git and Visual Studio (VS)

VS has Git support built in.

Features include:

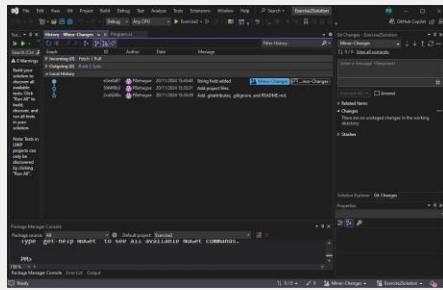
A Git Status Bar shows, branch, dirty indicators, incoming and outgoing commits

Can do most common Get operations from within the editor:

- Initialize a repository
- Clone
- Stage and Commit
- Push and Pull
- Create branches
- Resolve conflicts
- View diffs

Gain VS experience from Lab

QA



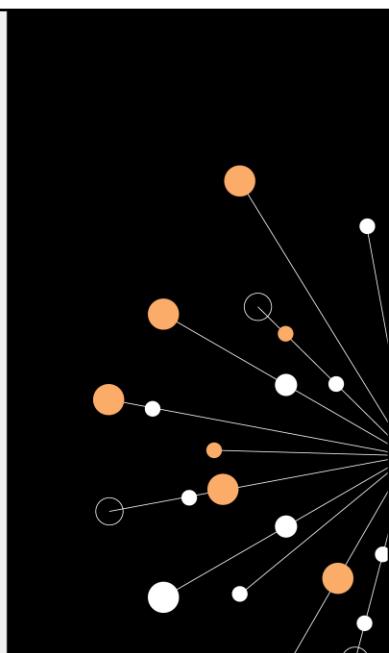
Hands on Lab

01 Using Visual Studio, Git and GitHub

Objective:

Your goal is to create and manage different versions of files in a GitHub repository using Git and Visual Studio Code

QA

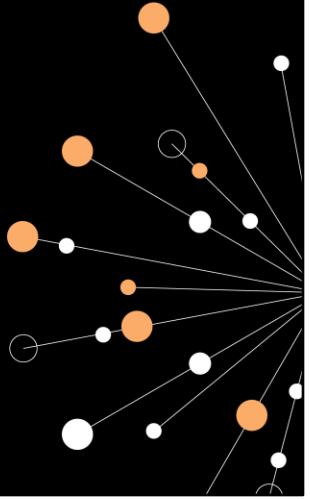


Summary

To understand the need for source control and how to use Git, GitHub and Visual Studio Code to manage a code repository.

- Introduction to Version Control
- Overview of Git and GitHub
- Installing and Configuring Git
- Basic Git Commands
- Introduction to GitHub
- Using Git with Visual Studio Code

QA



02 Object Oriented Programming Principles

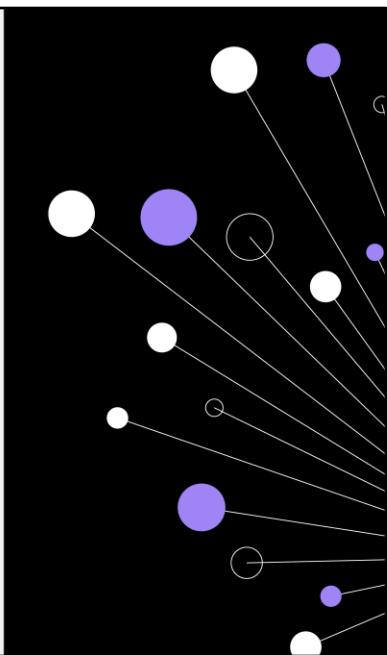
C# for HAAS F1



Learning objectives

- Review OOP basics (covered in C# Fundamentals course)
- Understand the core principles of Object-Oriented Programming (Encapsulation, Inheritance, Polymorphism, Abstraction)
- Gain an overview of constructors and initialisers
- Know how to create the different types of Property and when to use them
- Know how to populate collections with class instances

QA



Why Object Orientation?

Object-oriented programming (OOP) evolved from programming best practices.

Represents the real world

- People interact with *things*, not database records

Ease of maintenance

- Code is structured
- Functionality and data are together in one place
- Promotes code reuse through object instantiation or other OOP techniques, such as inheritance

C# is thoroughly object-oriented

- *Everything* is an object including value types



QA

Object orientation and the principles of object-oriented programming have been around for a long time. The approach evolved out of programming best practices and simply formalises those practices with language constructs.

A key aim of OOP is to represent the “real world”. In life, we don’t interact with data structures or database records, we interact with Customers, Employees, Products and Places to name just a few of the “objects” with which we might interact.

Another key aim of OOP is the ease of maintenance: we can write code in one place and modify that one copy of the code. Other parts of software which consume our code will carry on as before, but they will be using the modified code.

C# is a thoroughly object-oriented language.

Four Concepts

Encapsulation

- Keeping related data and methods together

Inheritance

- Hierarchical structure of objects, being able to inherit methods from a base (parent) class with no need to redeclare them

Polymorphism

- Sub-classing allows for methods in derived classes to override methods in the base (parent) class such that they have the same signature but behave differently

Abstraction

- The process of simplifying complex systems by focusing only on essential attributes and behaviours and hiding unnecessary details from the user/client.

QA



OO Fundamental – Abstraction (part of Design)

Ability to represent a complex problem in simple terms

- In OO, creation of a high-level definition with no detail
 - Detail added later in the process
- Factoring out common features of a category of data objects

Stresses ideas, qualities & properties not particulars

- Emphasises what an object is or does, rather than how it works
- Primary means of managing complexity in large programs

EXAMPLE:

Students (instances of type **Student**) attend a **Course**

- Have 'attributes'- name, experience, attendance record
- Have 'behaviour' - Listen(), Speak(), TakeBreak() DoPractical()
- Are part of 'relationships'
- A student 'sits on a' course, a course 'has' students

QA

OO Fundamental – Encapsulation

Hiding object's implementation, making it self-sufficient

Process of enclosing code needed to do one thing well

- Plus, all the data that code needs in a single object
- Allows complexity to be built from (apparently) simple objects
 - Internal representation & complexities are hidden in the objects
 - Users of an object know its required inputs and expected outputs
 - Substantial benefits in reliability , maintainability and re-use

Objects communicate via messaging (method calls)

- Messages allow (receiving) object to determine implementation
- Sender does not determine implementation for each instance

```
foreach(Student s in myStudents){ s.DoNextLab(30);}
```

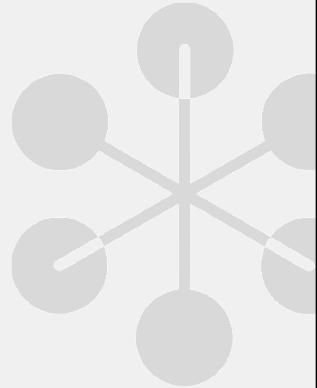
"I tell you how long you have, you sneak in the 'comfort' breaks"



QA

OO Fundamental - Inheritance

- **Inheritance** enables you to create new classes that reuse, extend, and modify the behaviour of other classes
- The class you inherit from is called the *base class* or *super class*
- The class that is being derived is called a *derived* or *sub class*
- Inheritance defines an '*is a kind of*' relationship
- In C#, you can only inherit from one base class
- A derived class can be a base class for another class that forms a transitive relationship
 - If ClassA is a base class and ClassB inherits from ClassA, ClassB is the derived class and inherits the members of ClassA
 - If ClassC inherits from ClassB , then ClassC inherits the members of ClassB and ClassA
- Derived classes do not inherit *constructors* or *finalizers*



QA

61

OO Fundamentals - Polymorphism

- **Polymorphism** is a Greek word meaning '*having many forms*'
- Polymorphism occurs because the *runtime* type of an object can be different to an object's *declared* type
- Objects of a *derived* type may be treated as objects of a *base* class in places, such as when passed as a parameter to a method, or when stored in a collection
- For example:
 - A **Rectangle** instance can be used anywhere a **Rectangle** type is expected
 - A **Rectangle** instance can be used anywhere a **Polygon** type is expected
 - A **Rectangle** instance can be used anywhere a **Shape** type is expected
 - A **Rectangle** instance can be used anywhere a **System.Object** type is expected



QA

62

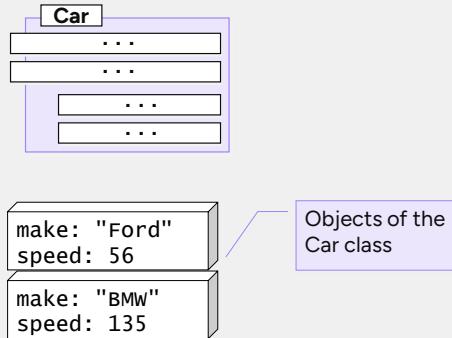
Classes and Objects

class definition is a blueprint, a 'plan' for making objects

- Architect can draw up plans, but you can't 'move in'
- Need to create an instance (a house) – then 'move in'

Objects are unique instances of a class

- Have their own *state* (values for their fields)
- Have their own *identity* (address in memory)
- Structure & behaviour of similar objects is defined by their class



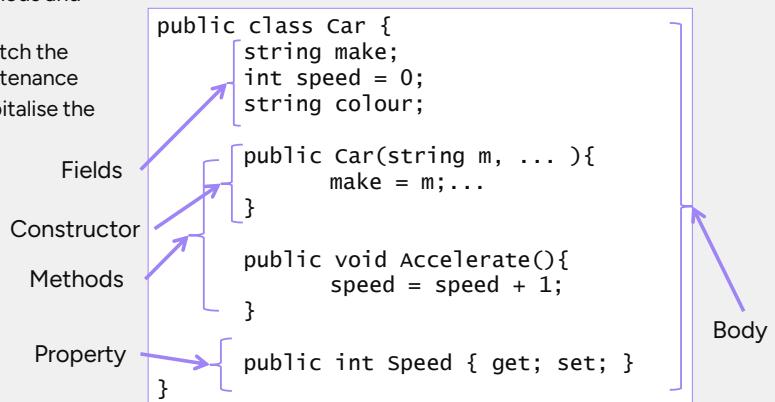
QA

Remember, the type definition is simply a blueprint of methods, fields and properties. Objects, on the other hand, are instances of the type. They have their own state (the values in the fields) and their own identity, which in the world of a computer program is the address in memory that the object is stored in.

C# Class

A C# class is a description of an object

- It contains the fields, methods and properties
- The class name should match the filename (for ease of maintenance)
- Naming convention is to capitalise the words in a name



QA

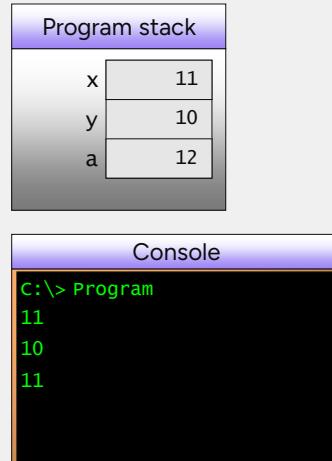
Classic Value Type Behaviour

```
public class Program {
    public static void Main() {
        int x = 10;
        int y = x;
        x++;
        Console.WriteLine(x);
        Console.WriteLine(y);
        Foo(x);
        Console.WriteLine(x);
    }

    public static void Foo(int a) {
        a = a + 1;
    }
}

public struct Int32 {...}
```

QA



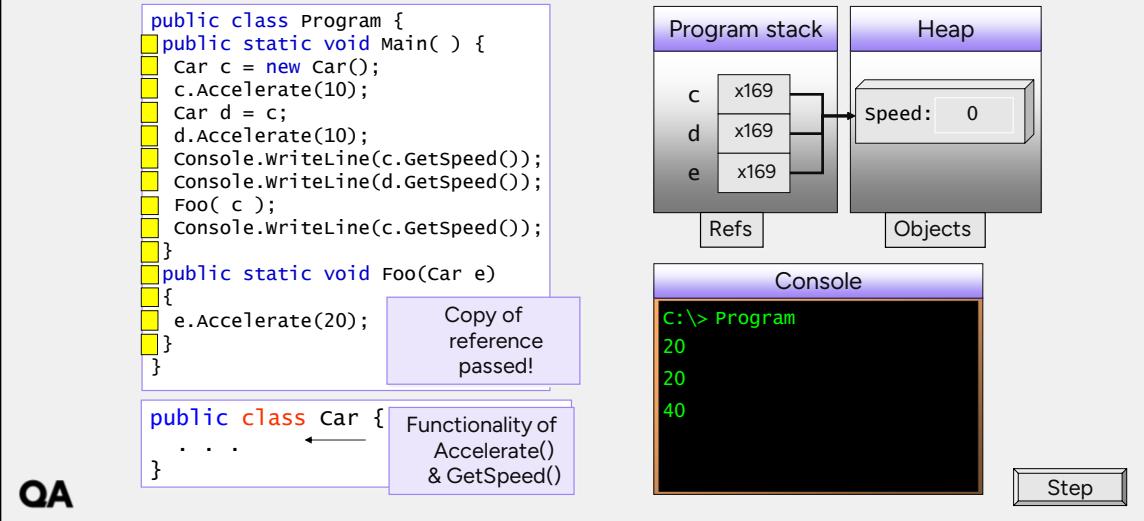
Value types behave very logically. Value types have their memory allocated in situ, which in the case of local variables means on the call stack. When you perform an assignment operation, the contents of the value type is copied to the destination variable, overwriting its contents.

This means that in the code above, when we declare the variable y and assign x into it, the contents of x (10 in this case) are copied into the storage for y. Any operations that are subsequently performed on x will therefore have no effect on the variable y.

Value types are named because of their "pass by value" semantics when they are passed as arguments to method calls. In the call to the method Foo(), a copy of the value type x is passed through to the method. This copy is then accessed using the symbolic variable name a. Operations performed on a have no effect on the variable x, because a is a copy of x. The copy is discarded (and the memory on the stack reclaimed) when the method Foo() returns.

You can see the output for the program in the console window on the right hand side above. Initially, x starts at 10. x is assigned into y (which will now also have the value 10). x is incremented, so now holds the value 11. The method Foo() is called, but x is unaffected by the call, so it still has the value of 11 at the end.

Reference Type Behaviour – different!



Reference types are a little bit more subtle than value types. A local variable of a reference type consists of a reference, which is on the stack. This variable refers to (in C++ they might have said points to) the actual object, which is allocated on the heap.

In our example above, `c` is a reference to a `Car` object that is allocated on the heap. Using this reference, we set the `Car` object's speed field to the value 10 (via acceleration). The code then declares another reference, `d`, into which `c` is assigned. It is here that the main behavioural difference between value and reference types starts to show up: only the reference is copied, not the object. Therefore, the result of the line of code

`Car d = c;`

is that there are now two references, `c` and `d`, both referring to the same object, as shown in the diagram below:

From this you can see that no matter which object, you will see changes when you access the `Car` object with the other reference.

It also follows that when `Foo()` is called, only a copy of the reference is passed to the method, so `e` also refers back to the same `Car` object on the heap. Therefore, any changes made using the reference `e` will be visible via reference `c` (as the changes occurred on the object referred to by `c`).

The null Reference

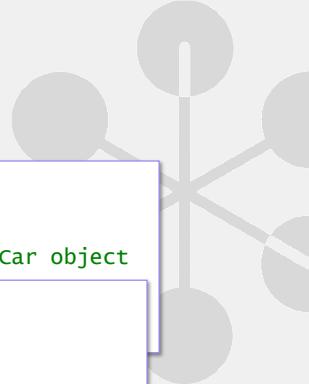
Variables of reference types may be set to null

- The variable does not reference an object
- If variable did reference an object, then old object is 'forgotten'

Can compare an object reference with null

```
public class QA {  
    public static Car FindPoolCar() {  
        Car aCar = null;  
        // attempt to make 'aCar' point to available Car object  
        return aCar;  
    }  
}  
  
Car car1 = QA.FindPoolCar();  
  
if( car1 != null ) {  
    // Drive the car away  
} else {  
    Console.WriteLine("No car available");  
}
```

QA



The default value for a variable whose type is a reference type (think class for now) is null. (provided it is not a local variable as then it is un-initialised like every local variable)

A variable 'theCar' above is initialised with a special reference called null, which indicates that the reference doesn't refer to any object. null is a keyword in the C# language, so you can use it with the equality operator to check whether an object reference has been initialised or not.

When you have finished using an object, you can set its object reference to null, but typically you just allow the reference to go out of scope. If there are no other references to the object, then the object becomes available for garbage collection.

null is meaningless for value types, as they have no reference capability.

Lists – revisited

All List variables are reference variables

```
public class Car {...} ← Assuming this class defined
```

```
Car[] cars1;
```

'cars1' is an un-initialised reference variable

```
List<Car> cars2 = new List<Car>();
```

'cars2' is a reference variable, .Count = 0
but contains no cars!!

```
List<Car> cars3 = new List<Car> { new Car(),  
new Car(),  
new Car()};
```

'cars3' - a reference to
a collection of car references

```
ProcessCarList(cars3);
```

QA

The sample method ProcessCarList(Car[] cars) would typically be void, it doesn't need to return anything as it can see (and change) the contents of the passed in collection.

"It is not possible to pass a List of cars...", "only a collection of car references" which the receiving code can iterate over, using either a foreach or a for loop using .Count.

Memory Management in the CLR

The CLR manages memory for you

- Reference types use memory allocated from the heaps
- CLR keeps track of references to objects

CLR uses a *garbage collector* to reclaim memory

- Background thread(s) that compact heap
- Object can be collected if no extant references exist
- Garbage only collected when necessary

Effects of CLR memory management on your code

- Very fast allocation strategy (much quicker than C++ heap)
- Cannot leak memory for objects
- Non-deterministic algorithm; you don't know when GC runs



QA

The CLR allocates memory from one of its heaps* for objects when you create them with new. Obviously, this only applies to reference types; value types are allocated memory directly on the stack. The CLR then tracks all of the references to the objects on the heap that exist within your code.

From time to time, the CLR will decide that it needs to reclaim some memory from the heap. It performs this task using a garbage collector that rapidly scans through the references and determines if any of the objects are unreachable from your code (i.e. all of the references to that object have gone out of scope). The garbage collector can then compact the heap reclaiming the memory from the objects that are no longer accessible.

To improve performance, .NET uses a multi-generational garbage collector. This means that it is very good at reclaiming memory from short lived objects, and doesn't bother to check the longer lived objects every time. It is therefore possible for an object to survive in memory for some time after the last reference to it has been removed, although it will get collected eventually.

What this all means is that memory allocation is very fast in .NET and that it is theoretically impossible to leak memory for a completely managed object (unless the CLR has a bug!). However, the downside of this is that you can never determine when an object will be removed from memory, nor when the GC

(which is self-tuning), will run.

* There is one heap for small objects and one for large objects (objects ~ 80Kb or larger).

Possible Problems with GC

Types might also require *guaranteed clean up*

- O/S files need to be closed to flush stream
- GDI+ handles have to be closed

Certain resources need to be released *deterministically*

- Database connections need to be closed to support pooling

CLR and FCL provide support for both

- Finalizer for guaranteed clean up
- Dispose pattern for deterministic resource release



QA

The use of a garbage collector (GC) poses some problems for us, particular when you start to use objects that encapsulate unmanaged resources, such as database connections, file handles, sockets and low level GDI or GDI+ objects. These objects use operating system resources that are written in native code, and the GC is unaware of pressure on these resources.

There are times when you need an object to guarantee clean up of an unmanaged resource. For example, it is all well and good reclaiming the memory from the heap for a File object, but you also need to ensure that the low level O/S file handle is closed (with the Win32 API CloseHandle), otherwise the content of the file will be lost! In these cases you must have code that will execute as part of the object's destruction process.

Let's consider another example where it is important to be able to release a resource in a deterministic manner. The SQL Server managed provider in ADO.NET supports connection pooling of low level (native) connections, with a default pool maximum of just 100 connections. Therefore, if you have no way to tell the pool that you had finished with the object, and you simply relied on the GC to release the resource, a busy ASP.NET application might soon run out of connections. In this case you need a way to tell the object to release its contained unmanaged connection back into the pool as soon as you have

finished with it, not when the GC runs.

Fortunately, the CLR and the FCL provide support for both of these requirements through the use of Finalizer methods and the Dispose Pattern.

The Finalizer Method

Purpose is to release resources owned by an object

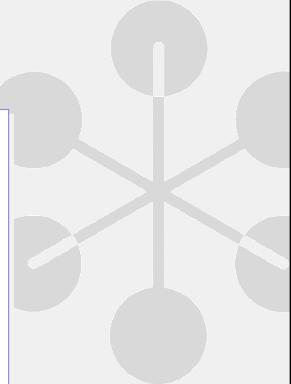
- Called by the GC except under truly disastrous circumstances
- Necessary if object *directly interacts* with unmanaged memory!

```
public class Font {  
    private Font() { ... }  
    public static Font CreateFont( string name, int size ) {  
        Font f = new Font();  
        f.handle = NativeCode.CreateFont( ... );  
        ...  
        return f;  
    }  
    ...  
    ~Font() {  
        NativeCode.SafeReleaseHandle( ref handle );  
    }  
    System.IntPtr handle = IntPtr.Zero;  
}
```

C# Finalizer - NOT a destructor

Code ends up in a try / finally

QA



In C#, a class can define a finalizer method which is used to clean up when the object dies. This is not a destructor (in the C++ sense of the word), as it can not be called deterministically, nor is it automatically called at predefined locations in code. Rather a garbage collector thread calls it when it needs to free up the memory of the object.

In C# the finalizer is written using a syntax that will be familiar to C++ developers. In reality this is actually an override of the Object.Finalize method, and you can check this out if you examine your code with ildasm.exe. Note that you can only override Object.Finalize using this syntax.

A finalizer should be used to free the additional resources held by the object, such as open files, window handles, database connections etc. C# manages memory automatically, so an object does not need to explicitly free up memory unless that memory was allocated by your object using unmanaged (native) APIs.

Dispose Method

Finalizer only called when object is garbage collected

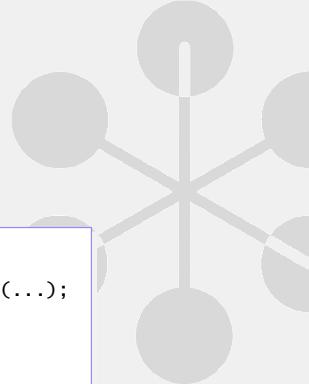
- GC only happens when managed memory is scarce
- GC is not triggered when other resources become scarce

Some resources must be explicitly released

- Provide a method for users to call, typically called `Dispose`
 - Other obvious names may be used, such as `Close`

```
public class Font {  
    public void Dispose() {  
        ...  
    }  
}
```

```
public class ChessBoard {  
    public void Show() {  
        Font f = Font.CreateFont(...);  
        ...  
        f.Dispose();  
    }  
}
```



QA

The finalizer method will be called automatically when the object's memory is garbage collected. Unfortunately, as we have already seen, there is no guarantee as to when this will happen or that it will happen before the program exits.

Clearly, this is unacceptable if resources are scarce. A manual mechanism has been added in the guise of a design pattern using the `IDisposable` interface and the `Dispose` method.

The class which encapsulates a resource should provide a method for disposing of the resource. The method could have any name, but for consistencies sake, and as we shall see later to enable the compiler to automatically generate calls, should be called `Dispose`. Certain resource types also tend to provide a `Close` method to perform the same task; developers somehow seem to be more comfortable closing a file rather than disposing of it!

Golden Rules:

- If a class has a finalizer it is generally considered to be good practice to provide a `Dispose` method.
- If a class has a `Dispose` method to release an unmanaged resource owned directly by that object then it must have a finalizer, as you can't rely on the developer that uses your class remembering to call `Dispose`.

Compiler Support for Dispose

Programmer has to remember to call `Dispose`

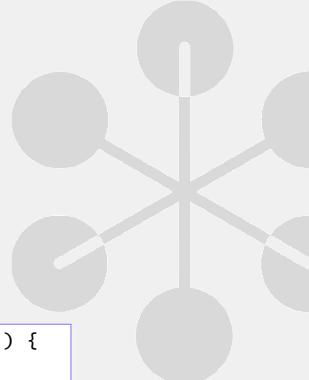
- Forgetting to do this can cause resource problems

The compiler can help

- The `using` statement ensures `Dispose` is called
- Irrespective of how the statement is exited, including by exception
 - Only for objects which implement `IDisposable`
 - No need to explicitly call `Dispose`

```
public class MyResourceHolder : IDisposable {  
    public void Dispose() { ... }  
}
```

```
using( MyResourceHolder mrh = new MyResourceHolder() ) {  
    ...  
}  
} Avoids nesting try / finally's
```



QA

It is very easy for a developer to forget to call `Dispose` on objects, especially when exceptions are being thrown. The only safe way to use such objects is to use them within a `try {} / finally {}` block, but this can get very tedious to code. To simplify the usage of classes that implement `Dispose`, the C# compiler can generate the `try {} / finally {}` code for you if you add a `using` statement block to surround the code where the object will be used.

Note that this use of `using` has nothing to do with namespaces.

The `using` statement defines a block of code, at the end of which the resources are no longer required. The compiler therefore ensures that the `Dispose` method gets called. The compiler needs to know that the object has a `Dispose` method, which it does by checking that the type implements the `IDisposable` interface.

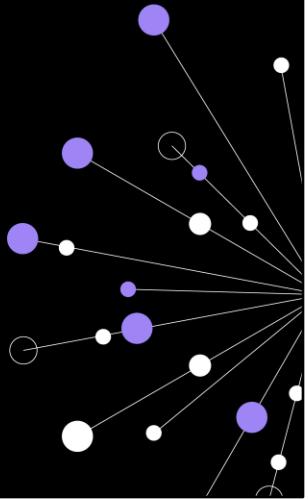
This interface only has a single method: `Dispose`.

Let's take a look at how you use `using` over the page.

Summary

- OO concept of defining a type
- Defining ref types – keyword class
- Understanding the concept of an 'object' reference
- Distinguishing between value and ref type behaviour

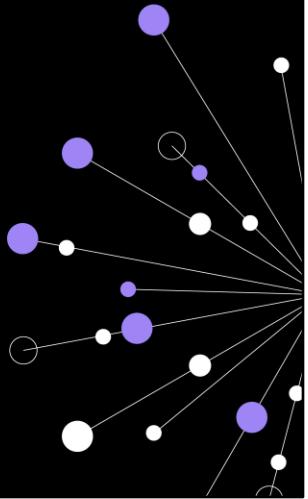
QA



Hands on Lab

Exercise 02 Object Oriented Programming Principles

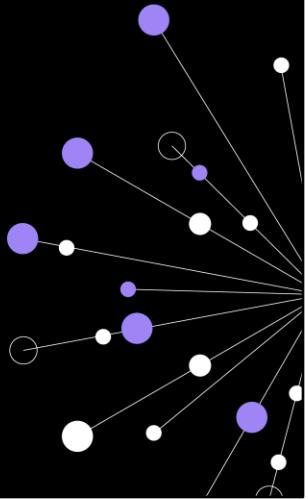
QA



Appendix

Parameter passing ("out" and "in")

QA



Passing Parameters: Value types By Reference 'OUT'

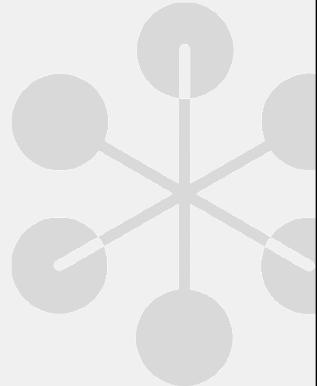
The value type variables are *passed by reference* using the **out** keyword in both the method declaration and the method call. Unlike **ref**, variables do not need to be initialised before being passed.

```
void OutExampleMethod(out int number, out string text, out string optionalString)
{
    number = 42;
    text = "I'm output text";
    optionalString = null;
}

int argNumber;
string argText, argOptionalString;
OutExampleMethod(out argNumber, out argText, out argOptionalString);

Console.WriteLine(argNumber);
Console.WriteLine(argText);
Console.WriteLine(argOptionalString == null);

// Output:
// 42
// I'm output text
// True
```



QA

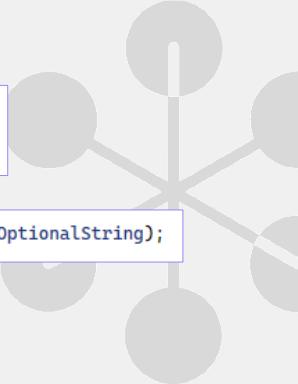
Use **out** parameters to output multiple values from a method.

Passing Parameters: Value types By Reference 'OUT'

From C# 7, you can declare the **out** variables in the argument list of the method call rather than having to declare them beforehand:

```
int argNumber;  
string argText, argOptionalString;  
OutExampleMethod(out argNumber, out argText, out argOptionalString);
```

```
OutExampleMethod(out int argNumber, out string argText, out string argOptionalString);
```



QA

From C# 7.0, you no longer need to declare the **out** variables outside of the method call. You can declare the **out** variables in the argument list of the method call.

Tuples instead of out

Out parameters are used to return multiple items from a method

An alternative is to return a **collection** when the values belong in a group of the same type e.g., `List<string>`

Another alternative is to return a **tuple**:

```
(int number, string text, string? optionalString) OutExampleMethod()
{
    var number = 42;
    var text = "I'm output text";
    string? optionalString = null;
    return (number, text, optionalString);

var outputs = OutExampleMethod();
Console.WriteLine($"Outputs: number is {outputs.number}, text is {outputs.text}");
Console.WriteLine($"Outputs: optional string is {outputs.optionalString ?? "NULL"}");
```

A **tuple** is concise syntax to group multiple data elements in a lightweight data structure.

The most common use case is as a method return type within private or internal utility methods.

QA



Passing Parameters: Value types as 'IN'

- The value type variable `x` is passed by reference using the `in` keyword
- `in` ensures the argument cannot be modified by the called method
- The `in` keyword is optional in the calling code because it is the default passing mechanism

```
PassingParams pp = new PassingParams();
int x = 5;
System.Console.WriteLine("The value before calling the method: {0}", x);
pp.SquareANumber(x); // Passing the variable by reference with 'in'
System.Console.WriteLine("The value after calling the method: {0}", x);

// Output:
// The value before calling the method: 5
// 25
// The value after calling the method: 5
```

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number * number);
    return;
}
```

QA

Passing Parameters: Value types As 'IN'

- The called method cannot modify the **in** parameter, so the code must be changed to reflect this restriction:

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number *= number);
    return;
}
```

[!] (parameter) **in** int number
CS8331: Cannot assign to variable 'in int' because it is a readonly variable

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number * number);
    return;
}
```

QA

The **in**, **ref**, and **out** keywords are not considered part of the method signature for the purpose of overload resolution.

Git Configurations (`git config`)

Need to have your username and email configured.

Can be done for single downloaded repository or globally for all cloned repositories

Setting Config Globally

```
# git config --global user.name "[USERNAME]"
$ git config --global user.name "John Doe"
# git config --global user.email "[EMAIL]"
$ git config --global user.email "johndoe@example.com"
```

Setting Config Locally:

```
# git config user.name "[USERNAME]"
$ git config user.name "John Doe"
# git config user.email "[EMAIL]"
$ git config user.email "johndoe@example.com"
```

QA



Git Configuration – Setting the User

Git keeps track of who performs version control actions

- Git must be configured with your own name and email

To configure Git with your name and email we use the following commands:

```
% git config --global user.name "Your Name"  
% git config --global user.email "mail@example.com"
```

The values can then be accessed using:

```
% git config user.xxx
```

You can list all the configurations with:

```
% git config --list
```



QA

02b LINQ

C# for HAAS F1



Learning objectives

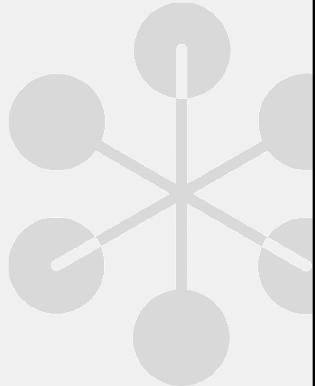
- Language-Integrated Query (LINQ)
- LINQ syntax
- LINQ projections
- Deferred execution
- Aggregations
- Forcing immediate execution
- Joins

QA



LINQ

- Language-Integrated Query (**LINQ**) is a set of technologies based on the integration of query capabilities directly into the C# language
- You can use LINQ to consistently query data from Objects, collections that support `IEnumerable`, relational databases and XML, all using C# syntax
- LINQ queries are written using *query expression syntax* or *method syntax*
- Some queries can only use the method syntax (e.g., `Count` and `Max`)
- A query is **not** executed until you iterate over the query variable



QA

Scenario: Non-LINQ And LINQ equivalent

Compare the two code snippets.

The left-hand snippet does **not** use LINQ.

The right-hand snippet uses LINQ.

```
// Non-LINQ Example  
  
// Specify the data source  
int[] scores = { 97, 92, 81, 60 };  
  
// Create a List to store the high scores  
List<int> highScores = new();  
  
// Iterate over the array to find the  
// scores above 80 and add to the List  
foreach (int score in scores)  
{  
    if(score > 80)  
    {  
        highScores.Add(score);  
    }  
}  
// Iterate over the highScores List  
foreach(int i in highScores)  
{  
    Console.WriteLine(i + " ");  
}  
// Output: 97 92 81
```

```
// LINQ  
  
// Specify the data source  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query  
foreach (int i in scoreQuery)  
{  
    Console.WriteLine(i + " ");  
}  
// Output: 97 92 81
```

QA

The two code snippets show a non-LINQ approach and a LINQ approach to the same scenario: finding scores that are above 80.

The LINQ snippet uses a query expression. The non-LINQ example requires a collection to store the results. The LINQ equivalent does not require the results to be stored. The LINQ query is not executed until the query variable is iterated over in the foreach loop.

LINQ Syntax

Query expression syntax:

```
// Specify the data source  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query  
foreach (int i in scoreQuery)  
{  
    Console.WriteLine(i + " ");  
}  
// Output: 97 92 81
```

Query method syntax:

```
// Specify the data source  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query using method syntax  
var query = scores.Where(score => score > 80)  
    .Select(score => score);  
  
// Execute the query  
foreach (var i in query)  
{  
    Console.WriteLine(i + " ");  
}  
// Output: 97 92 81
```

QA

The LINQ query expression syntax has been designed to be similar to Structured Query Language (SQL).

LINQ Syntax Example

Query expression syntax:

```
// Specify the data source
List<Customer> customers = Customer.GetCustomers();

// Define the query using query expression syntax
var queryExpression = from c in customers
    where c.City == "London"
    orderby c.Balance
    select c.CustomerName;

// Execute the query
foreach (var c in queryExpression)
{
    Console.WriteLine(c);
}
```

Query method syntax:

```
// Specify the data source
List<Customer> customers = Customer.GetCustomers();

// Define the query using method syntax
var queryMethod = customers.Where(c => c.City == "London")
    .OrderBy(c => c.Balance)
    .Select(c => c.CustomerName);

// Execute the query
foreach (var c in queryMethod)
{
    Console.WriteLine(c);
}
```

QA

Note the use of the **var** keyword. This provides flexibility in enabling you to project different types including anonymous types from your queries. Some LINQ queries project (return) an **IEnumerable** type, others return an **IQueryable** type. It is also possible to return a new anonymous type by selecting only a subset of attributes of an object to be returned.

LINQ Projections

You can explicitly specify the query type if you project (select) a whole class e.g. Customer.

```
// Return type is IEnumerable<Customer>
IEnumerable<Customer> queryA = from c in customers
where c.City == "London"
orderby c.Balance
select c; // Customer is returned
```

If you project only some of the object's properties, the compiler will generate a new anonymous type.

Use the **var** keyword:

```
// Return type is IEnumerable<<anonymous type: string customerName, string City>>
IEnumerable<Customer> queryB = from c in customers
where c.City == "London"
orderby c.Balance
select new { c.CustomerName, c.City }; // CustomerName and City are returned

// use var
var queryC = from c in customers
where c.City == "London"
orderby c.Balance
select new { c.CustomerName, c.City }; // CustomerName and City are returned
```

QA

Deferred Execution

A LINQ query is not executed until you iterate over the query variable in a foreach statement.

This allows the query to retrieve different data each time it is executed:

```
string[] names = { "Tommy", "Fiona", "Rashid", "Bobby" };

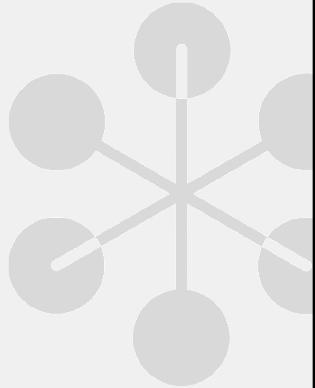
var query = from s in names
            where s.Length == 5
            select s;

foreach (string s in query)
{
    Console.WriteLine(s + " ");
}
// Output: Tommy Fiona Bobby

names[0] = "Susie";

foreach (string s in query)
{
    Console.WriteLine(s + " ");
}
// Output: Susie Fiona Bobby
```

QA



Aggregations

- Aggregations execute without an explicit foreach statement
- The **baseQuery** uses lambda expressions to specify which property of Customer is to be aggregated
- **Qry2** projects the Balance property which is a decimal, so no lambdas are required

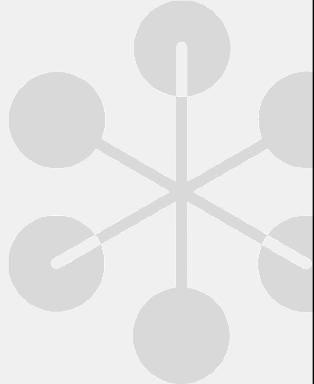
```
var baseQuery = from c in customers
                 where c.City == "London"
                 orderby c.Balance
                 select c;// Customer

decimal avg = baseQuery.Average(c => c.Balance);
decimal max = baseQuery.Max(c => c.Balance);
decimal total = baseQuery.Sum(c => c.Balance);

IEnumerable<decimal> qry2 = from c in customers
                               where c.City == "London"
                               select c.Balance;// Decimal

avg = qry2.Average();
max = qry2.Max();
total = qry2.Sum();
```

QA



Forcing immediate execution: Aggregate functions

- Queries that perform aggregation functions over a range of source elements must first iterate over those elements, therefore they execute **without** an explicit foreach statement
- These types of queries return a single value not an IEnumerable collection

```
int[] scores = { 97, 92, 81, 60, 40, 54, 80, 75 };

var failingScores =
    from score in scores
    where score < 80
    select score;

int countFailingScores = failingScores.Count();
Console.WriteLine("Number of failing scores is " + countFailingScores);

double avgFailingScores = failingScores.Average();
Console.WriteLine("Average of failing scores is " + avgFailingScores);
```

QA



Forcing immediate execution: `ToList` or `ToArray`

To force execution of any LINQ query and cache its results, you can call the `ToList` or `ToArray` methods:

```
string[] names = { "Tommy", "Fiona", "Rashid", "Bobby" };

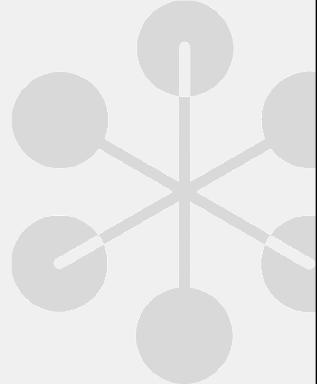
var query = (from s in names
             where s.Length == 5
             select s).ToList() //force execution

foreach (string s in query)
{
    Console.WriteLine(s);
}
// Output: Tommy Fiona Bobby

names[0] = "Susie";

foreach (string s in query)
{
    Console.WriteLine(s);
}
// Output: Tommy Fiona Bobby
```

QA



94

For applications that want to cache the results of query evaluation, two methods, `ToList` and `ToArray`, are provided that force the immediate evaluation of the query and return either a `List<T>` or an `Array` containing the results of the query evaluation.

Any changes to the underlying original collection would not be reflected when iterating over the `List`/`Array` now being used.

Both `ToArray` and `ToList` force immediate query evaluation. The same is true for any LINQ operations that return singleton values (for example: `First`, `ElementAt`, `Sum`, `Average`, `All`, `Any`).

Note that the method signature for `ToList` is:

`List<T> ToList<T>(IQueryable<T> source).`

You don't need to specify `ToList<string>()` in the above code because the compiler knows you are converting an `IEnumerable<string>` into a `List`.

Joins

The **join** clause is used to match elements in one collection with elements in another collection.

Find persons who have names that match the 'names' list:

```
string[] firstNames = { "Vinita", "Pete" };

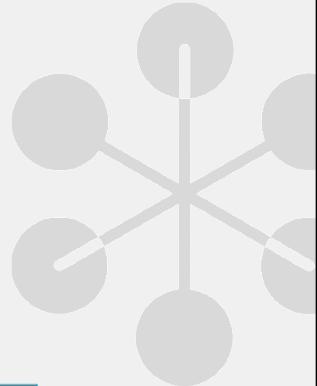
var people = new List<Person> {
    new Person("Pete", 20), new Person("Rafael", 50),
    new Person("Vinita", 32), new Person("Pete", 37),
    new Person("Tom", 40)
};

var matched = from n in firstNames
    join p in people
        on n equals p.Name
    select new { AgeOfPerson = p.Age, FirstName = n };

foreach(var p in matched)
{
    Console.WriteLine($"{p.FirstName} {p.AgeOfPerson}");
}
```

```
Vinita, 32
Pete, 20
Pete, 37
```

QA



95

We want to find elements of 'people' who are in the list 'names'.

The code on the slide could be simplified because the end result is simply a list of Person elements whose Name is in the list of names.

As all the information in the string array is effectively held in the Person array (because people have names) there is no need for the 'select' statement to select 'n' or any portion of it. Often when a join takes place the resulting select is likely to create instances of an anonymous class containing information from both 'n' and 'p' (i.e. both halves of the join).

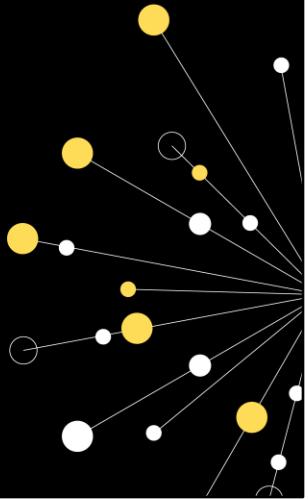
```
IEnumerable<Person> matched = from n in firstNames
    join p in people
        on n equals p.Name
    select p;

foreach(Person p in matched)
{
    Console.WriteLine($"{p.Name} {p.Age}");
}
```

Summary

- Language-Integrated Query (LINQ)
- LINQ syntax
- LINQ projections
- Deferred execution
- Aggregations
- Forcing immediate execution
- Joins

QA



Hands on Lab - OPTIONAL

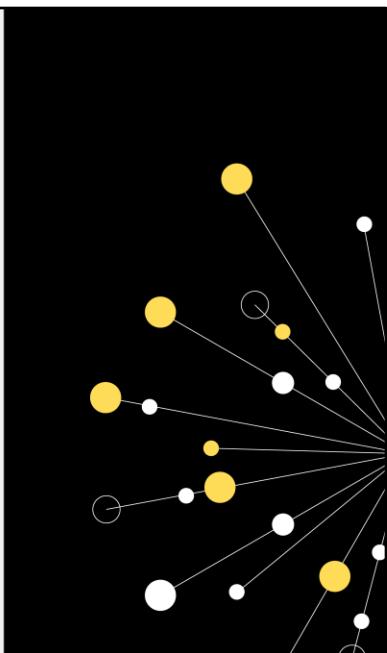
Exercise 02 LINQ - OPTIONAL

Convert non-LINQ code into the LINQ equivalent

Open the LINQLab starter project in C:\QAL\Sales Support\C#
for F1 HAAS\Labs\02b_LINQ\Begin

Reengineer code to use LINQ. Use hints use appropriate LINQ
functionality.

QA



O2c Methods, Properties & Constructors

C# for HAAS F1



Learning objectives

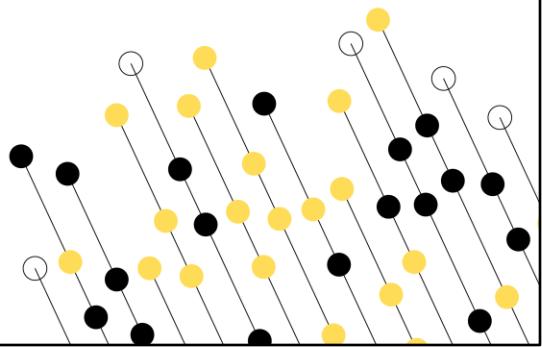
- Instance vs. Static
- Properties
 - Properties with backing fields
 - Auto-implemented properties
 - Calculated properties
 - Accessing properties
- Constructing objects
 - Constructors
 - Object initialisers
- The 'this' keyword

QA



Instance vs. Static

Methods, Properties and Constructors



Instantiating an object

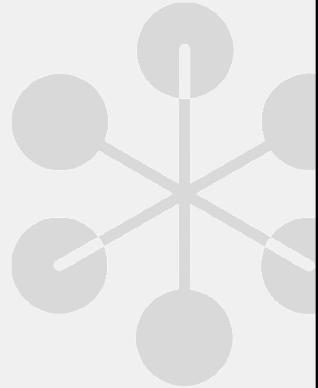
The class file is just the description of how the object is to be created and what it contains.

We need to instantiate the class and create an object before we can use it

- This uses the `new` keyword
- Calls the constructor in the class

```
class variableName = new Class();
```

There is only one class, but there can be many objects created from that class



QA

Instantiating an object

```
public class Car{  
    String make;  
    ...  
}
```



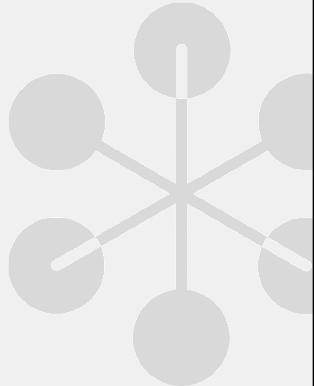
```
Car oldBanger = new Car();  
Car jalopy = new Car();  
Car buggie= new Car();  
Car wheels= new Car();
```



QA

Methods – a Reminder

- A method is a code block that contains a series of statements
- Methods are called (invoked) by a program
- Methods are declared in a *class*, *struct*, or *interface* with:
 - An access modifier such as **public** or **private**
 - Optional modifiers such as **abstract**
 - The return type such as **void** or **int**
 - The name of the method
 - Any method parameters in brackets
- This definition is known as the method signature
- Methods can be passed arguments that map to parameters defined in the method signature
- Every C# application has a method called **Main** that is the entry point for the application



QA

103

Note: In an application that uses top-level statements, the **Main** method is generated by the compiler.

Instance members – Methods, to exhibit functionality

Most methods operate on instances of a type

- Referred to as instance methods as each instance 'has' them
- Can't invoke them though unless you have an instance

Methods directly access instance variables

```
public class Car {  
    private int speed;  
  
    public void Accelerate( int weightOfFoot ) {  
        speed = speed + ...;  
    }  
    ...  
}
```

Field has 'class' scope but will belong to car 'objects'

No sign of keyword 'static'

QA

Methods are defined in a type and operate on instances of that type. Inside a method, you can declare local variables. Note that the scope of a local variable is limited to the block in which it is declared. This means that if you declare a local variable within a method, it cannot be accessed from outside of that method.

Invoking an Instance (non-static) Method

Use dot operator with a reference to the object

```
objectRef.MethodName(...)
```

```
public class Car {  
    public void Accelerate(  
        int weightOfFoot ) {  
        ...  
        ...  
    }
```

For *instance* methods, instantiate an object, then call the method using that object instance

Calls the Accelerate() method of the object referenced by car1

Car car1 = new Car();
car1.Accelerate(5); ✓
Car.Accelerate(5); ❌

Non static!!

QA

You use the dot operator to call an instance method of an object. Don't forget to include empty parentheses if the method has no arguments.

In the example on the slide, a new instance of the Car class is created. The Accelerate method is then called on that object, passing in 5 as the argument to represent how much acceleration the car should 'do'.

Static Methods

- Use the **static** modifier to declare a static member such as a class or method
- A **static** method belongs to the type itself rather than to a specific instance of the object
- Therefore, **static** methods do not require an object to be instantiated
- A **static** method can't be referenced through an instance
- **WriteLine** is an example of a **static** method of the **Console** class

```
Console.WriteLine();
↳ class System.Console
Represents the standard input, output, and error streams for console applications.
```



QA

106

A static class is a class that cannot be instantiated. You might want to write a class that only contains static methods or properties such as the framework-provided Math class.

A static class is restricted to having static members only so no instance methods, properties or fields are allowed.

Create and use a static method

- For *static* methods, use the **static** modifier on the method definition, then call the method using the class
- Issue a *using* directive to import the static members of the class to make the code less verbose

```
internal class PassingParams
{
    // reference
    public static void SquareANumber(int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
int x = 7;
PassingParams.SquareANumber(x); // call static method

// using static PassingParams
int y = 10;
SquareANumber(y); // call static method
```

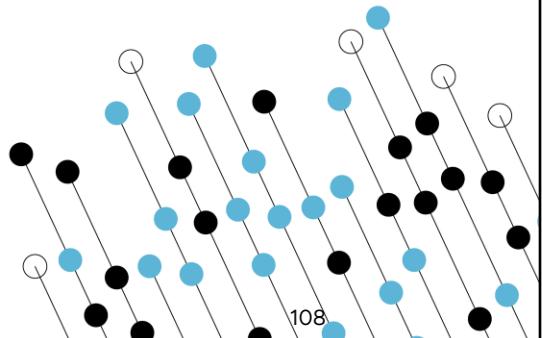
See Appendix for more examples

QA

107

Fields and Properties

Methods, Properties and Constructors



Fields

- A **field** is a variable that is declared directly in a class or struct
- A field can be an *instance* field: specific to the object instance
- A field can be a *static* field: shared amongst all objects of that type
- A best practice recommendation is to declare fields as **private** or **protected** and define *properties*

```
public class Car {  
    public string make;  
    public string model;  
    public int speed;  
}
```

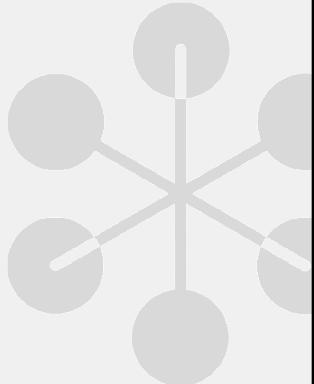
```
// fields do not provide validation beyond the datatype  
Car c1 = new();  
Car c2 = new();  
  
c1.make = "Ford";  
c2.make = "Ferrari";  
c1.model = "Fiesta";  
c2.model = "Flying submersible"; // not realistic  
c1.speed = 9999; // not realistic
```

QA

Properties

- **Properties** are special methods called *accessors*
- They provide a way to read, write, or compute the values of a private field whilst hiding the implementation
- A **get** property accessor is used to return the property value
- A **set** property accessor is used to assign a new value
- An **init** property accessor is used to assign a new value only during object construction
- Properties can be *read-write* (**get** and **set**)
- Properties can be *read-only* (**get**)
- Properties can be *write-only* (**set**)

QA



110

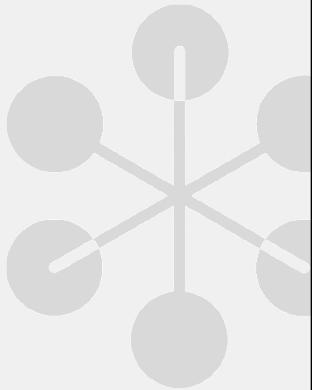
It is common to provide validation within the set accessor and perhaps conversion or computation within a get accessor.

The set accessor has a special parameter, called **value**.

Property Syntax

There are three different property syntaxes:

- Properties with backing fields
- Expression body definitions
- Auto-implemented properties



QA

111

Properties with backing fields

- The backing field holds the data
- The accessors can have different visibilities

```
public class Car
{
    private int speed;

    public int Speed
    {
        get { return speed; }
        private set { speed = value; }
    }
}
```

QA

112

The setter or the getter can be marked with a different visibility to the rest of the property to provide encapsulation. In the example, the get accessor (getter) for Speed is public whilst the set accessor (setter) is private. This means that code within the Car type can change its speed but no other code can.

Auto-implemented properties

- Use this syntax if there is no additional logic other than assigning or returning a value
- The C# compiler transparently creates the backing field for you and the implementation code to set / get to / from the backing field
- You can use an optional initialiser to set a value

```
public class Car
{
    0 references
    public string Make { get; init; } = "Ford";
    1 reference
    public int Speed { get; private set; } = 42;
}
```

Calculated Property Example

- **TempInDegreesCelsius** is a read-write auto-implemented property
- **TempInDegreesFahrenheit** is a calculated read-only property

```
public class Temperature
{
    public double TempInDegreesCelcius {get; set;}

    public double TempInDegreesFarenheit
    {
        get { return TempInDegreesCelcius * 1.8 + 32; }
    }
}
```

- **TempInDegreesFahrenheit** is a calculated read-only expression-bodied property

```
public class Temperature
{
    public double TempInDegreesCelcius {get; set;}

    public double TempInDegreesFarenheitAlt
        => TempInDegreesCelcius * 1.8 + 32;
}
```

QA

114

Accessing properties

Property access looks like field access to the client:

```
// instantiate a Car object instance
Car c1 = new();

// set a property value
c1.Model = "Xr2i";

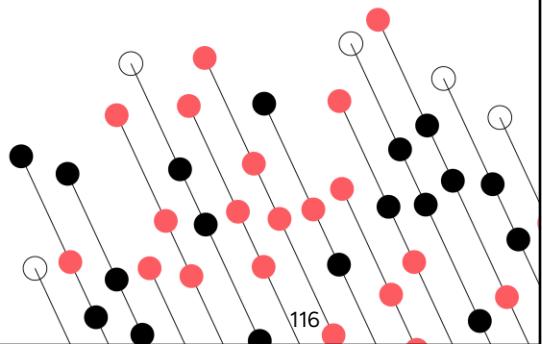
// get property values
Console.WriteLine(c1.Model);
Console.WriteLine(c1.Make);
Console.WriteLine(c1.Speed);
```

QA

115

Constructing Objects

Methods, Properties and Constructors



Object Construction

There are two ways to construct an object:

- Constructors
- Object initialisers

Constructors

- Define a constructor (or overloaded constructors) to include all combinations of *mandatory* fields
- This ensures the object is properly setup before being used

Object Initialisers

- Object initialisers can be used for *additional* optional fields that the object creator would like to set
- Object initialisers set properties or fields on the object *after* it has been constructed but before it is used



QA

Constructors

- A constructor is called whenever a class or struct is created
 - A constructor is a method whose name is the same as the name of its type
 - Constructors do not include a return type (not even void)
- Constructors are invoked using the **new** operator

```
Employee unknown = new(); //parameter-less constructor  
Employee spiderman = new("Peter", "Parker", 1); // 3 arg constructor
```

QA

118

Constructor Example: Setting properties

You have an employee class with three mandatory values

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }

    public Employee(string firstname,
                    string lastname,
                    int employeeID)
    {
        FirstName = firstname;
        LastName = lastname;
        EmployeeID = employeeID;
    }
}
```

[See Appendix for more ways to define constructors](#)

QA

119

The properties enable the 3 values to be accessed outside of the class. You also have the option of replacing the automatically implemented behaviour with your own code, if the need arises. For example, you may decide to output the **LastName** in uppercase.

Constructor Optional Parameters

A constructor is a method and can therefore specify optional parameters

```
public class Car
{
    public Car(string make="Ford",
               string model="Fiesta")
    {
        Make = make;
        Model = model;
    }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

[See Appendix for more ways to define constructors](#)

QA

120

Object Initialisers

- To avoid creating many overloaded constructors, *object initialisers* can be used to initialise an object into a ready state
- Object initialisers are often used for *optional* values and constructors are defined for *mandatory* values

```
// parameter-less constructor used implicitly with an object
initializer
Car c1 = new Car { Make = "Audi", Model = "TT", Speed = 70 };
Console.WriteLine($"Car {nameof(c1)} is make: {c1.Make} " +
    $"and model: {c1.Model} and Speed: {c1.Speed}");

// an explicit constructor can be used with an object initializer
Car c2 = new Car("Audi", "TT") { Speed = 70 };
Console.WriteLine($"Car {nameof(c2)} is make: {c2.Make} " +
    $"and model: {c2.Model} and Speed: {c2.Speed}");
```

QA

121

An object initialiser invokes the parameterless constructor unless an explicit constructor is specified

A struct always has a parameterless constructor

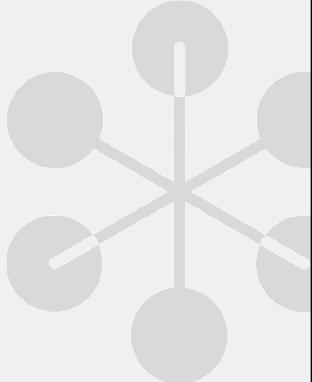
A class needs to have an explicit parameterless constructor defined if at least one explicit non-parameterless constructor is defined

Note: In C# 10 and later, you can explicitly declare a parameterless constructor in a struct. In C# 9.0 and earlier, the compiler always produced an implicit parameterless constructor that initialized all fields and properties to their default value. For example, **0** (zero) for numeric types, **false** for bools and **null** for reference types.

Where should our 'new' type be defined?

So far – largely working with Console Applications

- Compile into .exe's
- Coding mostly in 'Main' method of class Program (Occasionally invoking other helper methods of your class)
- Where in Visual Studio do we define class Car, class Person etc?
- If they are in a .exe, are they reusable functionality – no!



Welcome to class libraries (dll's)

- Project type 'Class Library' (in new project dialog)
- The FCL is a large bunch of .dll's (you add new dll's to these)
- Are they all magically visible?
- No, you need a reference to a .dll to 'consume' its types

QA

Before considering visibility modifiers like public, private we need to think first about where a type will 'live'

If we just stick class Car in file Car.cs alongside class Program (with its entry point Main) of Program.cs then the compiled code (the IL) of Car is now living in a .exe file.

Is this class reusable outside this .exe? i.e. Can another .exe use this .exe to use Car?

No

Car and maybe Train and Scooter etc perhaps defined in a xxx.xxx.Vehicles namespace should live in a project of type Class Library that compiles into a .dll. An exe, any exe, multiple exe's can then reference and use that dll and 'consume' its types.

Lets see how.

Using types defined in other Assemblies

Need a reference to the other Assembly

- Right click for 'Add Reference' dialog

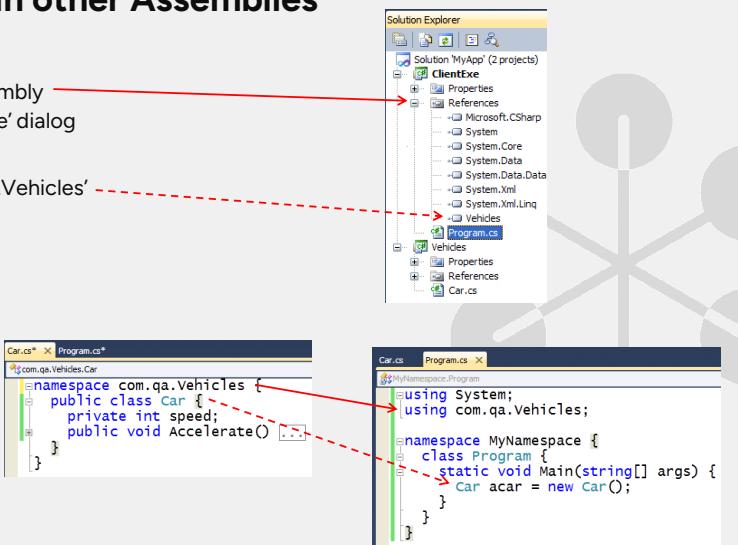
Define your class - 'Car'

- In your namespace - 'com.qa.Vehicles'

using statement

- Enables easy use
- Use last part of name only

QA



For a client 'exe' application to consume types (classes) defined in other .NET Assemblies (physical files - 'dll's) then the Client project needs a 'reference' to the 'library' assembly.

In the example above class 'Car' is defined in the Vehicles project that compiles into Vehicles.dll. But 'Car' is defined in the 'com.qa.Vehicles' namespace. This means its full name is com.qa.Vehicles.Car.

By right-clicking References in the ClientExe project an 'Add Reference' dialog appears allowing selection of any installed framework .NET dll and also any Class Library project in the open solution.

Only when a reference exists to a library can you start to use 'using' any namespaces defined in that library.

In the example above the 'using com.qa.Vehicles;' statement enables the Intellisense to show 'Car' as a visible datatype when writing code in the 'Main' method of class 'Program' in the ClientExe Assembly that has a reference to the Vehicles.dll (where com.qa.Vehicles.Car is defined).

The 'this' keyword

The **this** keyword has many uses:

- To qualify members hidden by similar names
- To pass an object as a parameter to other methods
- To chain constructors

```
class Employee
{
    private readonly string firstName;

    0 references
    public Employee(string firstName)
    {
        this.firstName = firstName;
    }
}
```

```
public Employee()
{
    BookSeatInTheOffice(this);
}
```

```
// parameter-less constructor
1 reference
public Car() :this("Unknown")
{ }

// 1 arg constructor
2 references
public Car(string make) : this(make, "Unknown model")
{ }
```

QA

124

In an 'instance' context there is always a 'this'

It is a reference to the object on which the method was invoked.

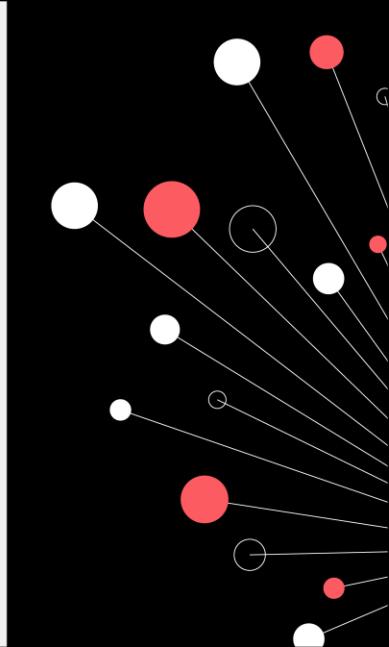
It is often referred to as the 'hidden' first parameter (of an instance method).

Note: the **this** keyword is also used to define *indexers* and as a modifier of the first parameter of an *extension method*.

Learning objectives

- Instance vs. Static
- Properties
 - Properties with backing fields
 - Auto-implemented properties
 - Calculated properties
 - Accessing properties
- Constructing objects
 - Constructors
 - Object initialisers
- The 'this' keyword

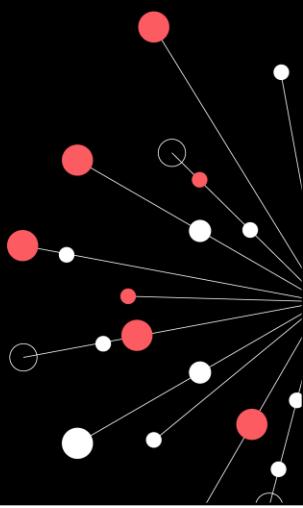
QA



Hands on Lab

Exercise 02 Review of OOP

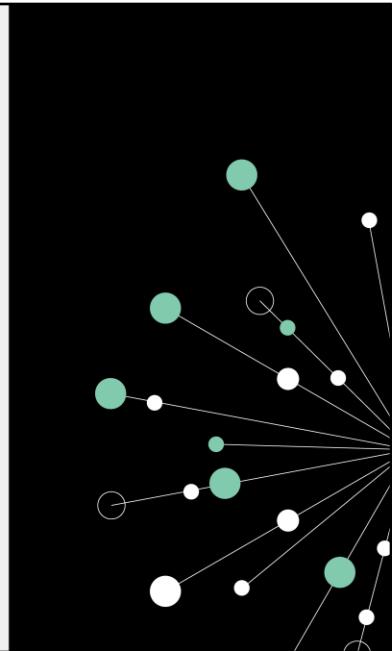
QA



APPENDIX

- Static Methods examples
- Expression Bodied Properties
- Constructor Examples

QA



Static Methods: Example

- The System.Math class provides constants and static methods for common mathematical functions
- The directive using System; is used in this example

```
int x = 5;
int y = 7;

int lowest = Math.Min(x, y);
int highest = Math.Max(x, y);

Console.WriteLine($"The lowest value is {lowest}");
Console.WriteLine($"The highest value is {highest}");
/* Output:
 * The lowest value is 5
 * The highest value is 7
 */

double price = 9.99;
double priceFloor = Math.Floor(price);
double priceRounded = Math.Round(price, 0);

Console.WriteLine(priceFloor); // 9
Console.WriteLine(priceRounded); // 10
```

QA

128

The **System.Math** class provides constants and static methods for common mathematical functions. This example has a using directive:

using System;

The static methods in the Math class must be prefixed with the class name.

Static Methods: Example 'using static'

- The directive **using static System.Math;** is used in this example

```
// using static System.Math
int x = 5;
int y = 7;

int lowest = Min(x, y);
int highest = Max(x, y);

Console.WriteLine($"The lowest value is {lowest}");
Console.WriteLine($"The highest value is {highest}");
/* Output:
 * The lowest value is 5
 * The highest value is 7
 */

double price = 9.99;
double priceFloor = Floor(price);
double priceRounded = Round(price, 0);

Console.WriteLine(priceFloor); // 9
Console.WriteLine(priceRounded); // 10
```

QA

129

This example has a using directive:

using static System.Math;

The static methods in the Math class do not need to be prefixed with the class name.

The 'using static' directive operates on a class rather than a namespace. It puts all the static members of that class directly into scope.

Expression bodied properties

- Property accessors often consist of single-line statements that just assign or return the result of an expression
- Any single-line expression can be implemented using expression body syntax
- If the property is *read-only* (**get**) you can omit the **get** keyword
- If the property is *read-write* (**get** and **set**) you must use the **get** and **set** keywords

QA

```
public class Car
{
    private int speed;
}
```

```
public class Car
{
    private int speed;

    public int Speed
    {
        get => speed;
        private set => speed = value;
    }
}
```

130

You omit the **return** keyword in the getter when using expression body syntax.

Constructor Example: Employee with Read-only Fields

You have an employee class with three mandatory values that should not be able to be changed once set: **firstName**, **lastName**, and **employeeID**.

Option 1

Define *readonly* fields and set their values in the constructor

```
class Employee
{
    private readonly string firstName;
    private readonly string lastName;
    private readonly int employeeID;

    0 references
    public Employee(string firstName, string lastName, int employeeID)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.employeeID = employeeID;
    }
}
```

this refers to the object on which field belongs

QA

131

The **readonly** modifier on the fields means that the fields' values can only be set in the constructor. Once the construction of an instance of the class is completed, the data within that instance's fields cannot be changed.

Fields do not provide any validation or conversion capabilities.

The fields are private and therefore not accessible outside of the class unless you provide a method to make these values readable. Typically, you would achieve this using the **ToString** method.

The **this** keyword disambiguates the parameter name **firstName** to the field belonging to the object instance **this.firstName**

Note: You can create a constant field using the **const** keyword. A constant must be initialized at the point of declaration whereas a **readonly** field can either be initialized at the point of declaration or in a constructor.

A **const** is a *compile-time* constant. A **readonly** field is a *run-time* constant.

Constructor Example: Auto-implemented properties

You have an employee class with three mandatory values that should not be able to be changed once set: **firstName**, **lastName**, and **employeeID**.

Option 2

Define *auto-implemented properties* with **get** accessors only.

```
class Employee
{
    public Employee(string firstName, string lastName, int employeeID)
    {
        FirstName = firstName;
        LastName = lastName;
        EmployeeID = employeeID;
    }

    // auto-implemented properties
    public string FirstName { get; }
    public string LastName { get; }
    public int EmployeeID { get; }
}
```

QA

132

The read-only properties enable the 3 values to be accessed outside of the class. You also have the option of replacing the automatically implemented behaviour with your own code, if the need arises. For example, you may decide to output the **LastName** in uppercase.

Constructor Example: Employee with full properties

Option 3

Define *full properties* with **get** and **init** accessors.

QA

133

```
class Employee
{
    1 reference
    public Employee(string firstName, string lastName, int employeeID)
    {
        FirstName = firstName;
        LastName = lastName;
        EmployeeID = employeeID;
    }
    // full properties with backing fields
    2 references
    public string FirstName
    {
        get { return firstName; }
        init
        {
            if (value.Length > 0)
            {
                firstName = value;
            }
        }
    }
    2 references
    public string LastName
    {
        get { return lastName.ToUpper(); }
        init
        {
            if (value.Length > 0)
            {
                lastName = value;
            }
        }
    }
    // backing fields
    private string firstName;
    private string lastName;

    // auto-implemented readonly property
    1 reference
    public int EmployeeID { get; }
}
```

You have an employee class with 3 mandatory values that should not be able to be changed once set: **firstName**, **lastName** and **employeeID**.

The **init** accessor of the full property enables validation logic to be used whilst restricting the setting of the property to construction time.

The **get** accessor of the full property enables conversion logic to be used whilst reading the property value. The property value is visible outside of the class.

Expression bodied Constructors

If a constructor can be implemented as a single statement, you can use an expression body definition:

```
public class Location
{
    // private backing field
    private string locationName;

    // expression bodied constructor
    public Location(string name) => Name = name;

    // expression bodied property accessors
    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

```
Location home = new("Home");
Location work = new("Work");
Console.WriteLine(home.Name);
Console.WriteLine(work.Name);
```

QA

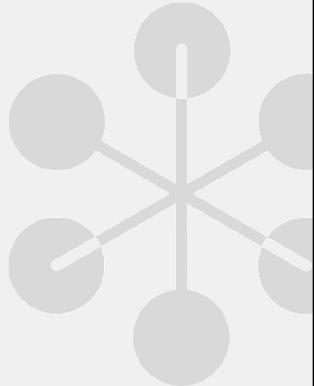
134

Constructor access modifiers

Constructors can be marked as

- public
- private
- internal
- protected internal
- private protected

These define how users of the class can construct an instance of the class.



QA

135

Constructor Overloading

A constructor is a method and can therefore be overloaded:

```
public class Car
{
    // parameter-less constructor
    1 reference
    public Car()
    {
        Make = "Unknown";
    }
    // 1 arg constructor
    0 references
    public Car(string make)
    {
        Make = make;
    }
    // 2 arg constructor
    0 references
    public Car(string make, string model)
    {
        Make = make;
        Model = model;
    }
}
```

QA

136

Constructor chaining

A constructor can invoke another constructor in the same object (constructor chaining) to avoid duplicating code, using the **this** keyword:

```
public class Car
{
    // parameter-less constructor
    public Car() :this("Unknown")
    {}

    // 1 arg constructor
    public Car(string make) : this(make, "Unknown model")
    {}

    // 2 arg constructor
    public Car(string make, string model)
    {
        Make = make;
        Model = model;
    }
}
```

```
Car c1 = new();
Car c2 = new("Audi");
Car c3 = new("BMW", "X5");

Console.WriteLine($"Car {nameof(c1)} is make: {c1.Make} and model: {c1.Model}");
Console.WriteLine($"Car {nameof(c2)} is make: {c2.Make} and model: {c2.Model}");
Console.WriteLine($"Car {nameof(c3)} is make: {c3.Make} and model: {c3.Model}");

// Output:
// Car c1 is make: Unknown and model: Unknown model
// Car c2 is make: Audi and model: Unknown model
// Car c3 is make: BMW and model: X5
```

QA

137

this can be used with or without parameters and any parameters in the current constructor are available to be passed to the called (chained) constructor.

This is used to avoid repeating / duplicating code in different overloaded constructors. It is typical for the constructors with the fewest parameters to call constructors with the most parameters.

Static Constructors

- Constructors can be marked as **static**
- A **static** constructor *initialises* any *static data* or performs an *action* that needs to be performed only once
- Static constructors do not have an *access modifier* or *parameters*
- Any class or struct can only have *one* static constructor
- Static constructors therefore **cannot** be *overloaded*
- Static constructors are called *automatically* by the CLR

```
class Employee
{
    static readonly string companyName;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static Employee()
    {
        companyName = "QA Ltd";
    }
}
```

03 Inheritance and Abstract Classes

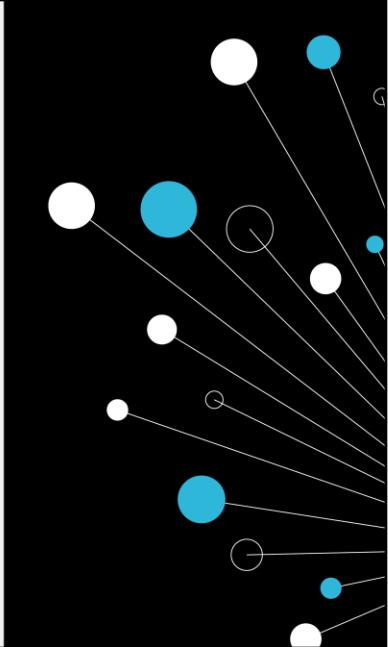
C# for HAAS F1



Learning objectives

- Inheritance
- Derived constructors
- Polymorphism
- Virtual members and overriding
- Invoking base class functionality
- Abstract classes
- Casting types: Up-casting, down-casting, is and as operators
- Overriding System.Object methods
- Sealed classes and members

QA



Inheritance

- Inheritance enables you to create new classes that reuse, extend, and modify the behaviour of other classes
- The class you inherit from is called the *base class* or *super class*
- The class that is being derived is called a *derived* or *sub class*
- Inheritance defines an *'is a kind of'* relationship
- In C#, you can only inherit from one base class
- A derived class can be a base class for another class that forms a transitive relationship
 - If ClassA is a base class and ClassB inherits from ClassA, ClassB is the derived class and inherits the members of ClassA
 - If ClassC inherits from ClassB , then ClassC inherits the members of ClassB and ClassA
- Derived classes do not inherit *constructors* or *finalizers*



QA

Examples of inheritance are ubiquitous in our model of the real world. Our definition of a cat inherits the features of our definition of a mammal, which in turn inherits the features of our definition of an animal. A chair and a table both inherit the features of furniture. A taxi is a kind of car.

Notice that derived classes take on features of the base class and can add or modify features, but they cannot remove features.

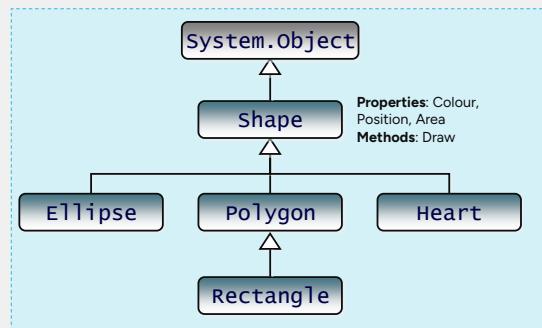
Note: A finalizer is used to perform any necessary clean-up when an object instance is being garbage collected.

Inheritance pertains to classes and not to structs.

Inheritance example: Graphics Application

Scenario: We need to be able to draw different shapes in our graphics application.

- Different shapes have common **properties**. Each shape needs to be filled with a *colour* and has a *position* and an *area*
- Different shapes have common **behaviours**. Each shape needs to be able to *draw* itself



QA

An example where inheritance might be a useful tool is a graphics application that needs to draw shapes. Such a program would allow us to create common shapes, such as ellipses, rectangles and triangles. Each of these is likely to be a class in the application's model. However, it quickly becomes apparent that these classes all share common behaviours and properties; shapes are filled with a colour, all need to be drawn, etc.

Good OO practice would have us factor the common elements into a base class, **Shape**, from which all of the other classes would be derived. These derived classes would then gain the benefit of re-use of the code from the base class. However, the derived types would also need to be able to extend and modify the base class functionality; for example, each separate type would need to be able to provide its own algorithm to calculate the area of the shape, and many would need to add specific fields and constructors to support their different data requirements.

One of the key things that we observe is that we can apply a test to see whether inheritance will work in our model: the "is a kind of" relationship test. A triangle is a kind of shape; an ellipse is a kind of shape; a circle is a kind of ellipse. This test confirms that we are introducing inheritance relationships that make logical sense.

Any class that does not explicitly extend another class implicitly extends the **System.Object** class. The Object class is the only class that does not have a base class and is the root class of all other classes.

Single inheritance means that each class can only have one direct *base class*: the direct base class of Circle is Ellipse; the direct base class of Ellipse is Shape.

Inheritance example: Graphics Application

Declare the base class: **Shape**

```
public class Shape {  
    public Color Colour { get; set; }  
    public Point Position { get; set; }  
    //other Shape properties and methods  
}
```

Define the derived class: **Polygon**

Use **derived : base** to specify an inheritance relationship

Add additional properties and methods as required

```
public class Polygon : Shape {  
    public int NumberOfSides { get; set; }  
}
```



QA

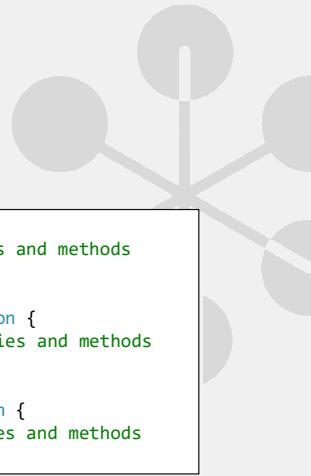
Inheritance example: Graphics Application

- A **Polygon** is a kind of **Shape**
- An **Ellipse** is a kind of **Shape**
- A **Rectangle** is a kind of **Polygon** and a kind of **Shape**
- A **Triangle** is a kind of **Polygon** and a kind of **Shape**

```
public class Polygon : Shape {  
    public int NumberOfSides { get; set; }  
}
```

```
public class Ellipse : Shape {  
    //ellipse-specific properties and methods  
}  
  
public class Rectangle : Polygon {  
    //rectangle-specific properties and methods  
}  
  
public class Triangle : Polygon {  
    //triangle-specific properties and methods  
}
```

QA



In a derived class, you only need to provide code for the things which are different to the base class.

Use a colon after the derived class name followed by the base class you wish to inherit from.

Note: if a base class is omitted, the compiler will implicitly inherit from the base class **System.Object**.

Derived Constructors

```
public class Shape
{
    1 reference
    public Point Position { get; set; }
    1 reference
    public Color Colour { get; set; }
    //The only way to instantiate a Shape is to specify a colour and position
    //This is still true for derived classes
    1 reference
    public Shape(Point position, Color colour)
    {
        Position = position; Colour = colour;
    }
}

public class Ellipse : Shape
{
    1 reference
    public int XRadius { get; set; }
    1 reference
    public int YRadius { get; set; }
    2 references
    public Ellipse(Point position, Color colour, int xRadius, int yRadius)
        : base(position, colour) //chain to base class constructor
    {
        XRadius = xRadius;
        YRadius = yRadius;
    }
}

Ellipse e1 = new Ellipse(new Point(4, 7), Color.Azure, 23, 34);

Ellipse e2 = new(new(4, 7), Color.Azure, 23, 34);
```

QA

In the example the Shape class only has one constructor which takes two parameters: position and colour.

Constructors are not inherited. But the rules they encapsulate are. Any class derived from Shape, *must have* some way of telling its base class what position and colour it should have.

The Ellipse constructor takes four parameters, of which the first two (position and colour) are passed up to the base class constructor. To invoke a constructor in the base class, use the *base* keyword. To invoke a constructor within the same class, use the *this* keyword.

It isn't possible to use both *this* and *base* in the same constructor; it's an either / or situation.

Polymorphism

- **Polymorphism** is a Greek word meaning '*having many forms*'
- Polymorphism occurs because the *runtime* type of an object can be different to an object's *declared* type
- Objects of a *derived* type may be treated as objects of a *base* class in places, such as when passed as a parameter to a method, or when stored in a collection
- For example:
 - A **Rectangle** instance can be used anywhere a **Rectangle** type is expected
 - A **Rectangle** instance can be used anywhere a **Polygon** type is expected
 - A **Rectangle** instance can be used anywhere a **Shape** type is expected
 - A **Rectangle** instance can be used anywhere a **System.Object** type is expected



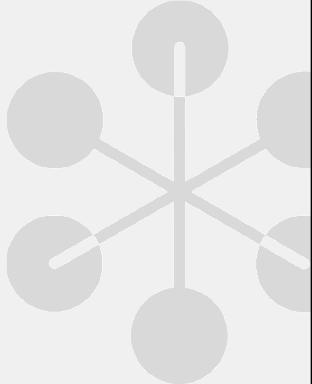
QA

Polymorphism Scenario

- The **Drawing** class needs to hold a collection of **Shapes** and be able to iterate over the collection to call each shape's *Draw* method

```
//A drawing has a collection of Shapes
public class Drawing
{
    private List<Shape> shapes;
    public List<Shape> Shapes
    {
        get
        {
            // null-coalescing assignment operator
            shapes ??= new List<Shape>();
            return shapes;
        }
    }
    public void Draw(Graphics canvas)
    {
        foreach (Shape shape in Shapes)
        {
            shape.Draw(canvas);
            // all shapes must have a Draw method
        }
    }
}
```

QA



In our drawing program, it seems likely that a collection of all of the different shaped objects will be maintained in some kind of a drawing type.

This collection would hold references to Rectangle, Ellipse and Triangle objects, but the code above treats them all as if they were mere Shape objects when it iterates through the collection to draw all the shapes. This generalised approach to working with objects is made even more powerful because of the fact that we can access behaviours of objects polymorphically.

When you generalise a **Rectangle** as a **Shape**, polymorphism will call Rectangle-specific methods rather than general Shape methods, if they exist.

Polymorphism with virtual methods or properties

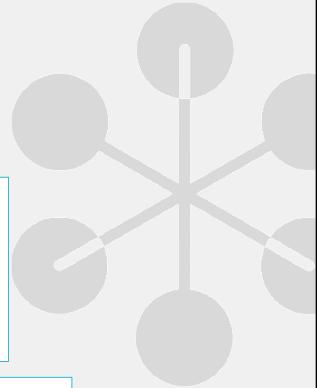
- Inherited methods and properties can be defined as **virtual**
- **Virtual** members can be **overridden** in the derived class
- This enables you to *generalise* the type of an object to its base class type, but have the compiler call the more *specialised* derived version of the member

```
public class Shape
{
    public virtual int Area
    { get; }
}
```

```
public class Ellipse : Shape
{
    public override int Area
    {
        get;
    }
}
```

```
Ellipse e = new Ellipse();
Shape s = e; // runtime type is Ellipse, declared type is Shape
Console.WriteLine(s.Area); // polymorphically gets Ellipse Area not Shape Area
```

QA



A derived class inherits all of the instance methods and properties of its base class. However, it can also modify inherited behaviour by overriding it. This means that the derived class defines a method or property with exactly the same signature and return type as one in a base class (not necessarily its immediate base class).

In C#, a class must explicitly allow a method to be overridden through the use of the “virtual” keyword. Otherwise, an attempt to override will generate a compiler warning.

C# will call the runtime’s specialized member rather than the declared type’s generalized member. This is polymorphism: treating objects generally but having them act specifically.

Member access modifiers

Members (methods and properties) can be marked as:

- public
- private
- protected
- internal
- protected internal
- private protected

These define how users of the class or a derived class can access the members of that class.



QA

A “public” member: access is not restricted. Many classes are marked with the public keyword, as are methods and properties that represent the publicly accessible façade of a type.

A “private” member: access is limited to the containing type. This modifier is very commonly applied to fields and occasionally to some methods and properties.

A “protected” member: access is limited to the containing class or types derived from the containing class.

An “internal” member: access is limited to the current assembly. Certain classes, known as helper classes, and some methods and properties will be specified with internal access.

A “protected internal” member: access is limited to the current assembly or types derived from the containing class.

A “private protected” member is accessible by types deriving from the containing class, but only within its containing assembly.

Invoking base class functionality

- A **derived** class can access **base** class members
- This avoids code duplication and having to have access to private fields
- To call a **base** class member, use the **base** keyword
- This calls the first matching member in the inheritance hierarchy

```
public class Shape
{
    2 references
    public virtual void Draw() { }
    1 reference
    public virtual void Draw(Graphics canvas) { }
    1 reference
    public Color Colour { get; set; }
    2 references
    public virtual int Area
    { get; }
```

```
public class Ellipse : Shape
{
    2 references
    public override void Draw()
    {
        base.Draw(); // invoke base class method first, Shape.Draw()
        Brush br = new SolidBrush(base.Colour); // perform additional functionality
                                                // using the Colour property from the base class
                                                // Shape.Colour
    }
}
```

QA

Common overriding scenarios are:

- Replace the virtual method's code completely
- Call the base class's implementation first, then add extra code
- Perform some of your own code, then call the base class's code

To do this, you use the **base** keyword and call the appropriate method or property. This will call the matching method in the base class, or in one of its base classes if the method is virtual and hasn't been overridden in the immediate base class.

You cannot call “*base.base*”. The **base** keyword can only be used to refer to the immediate base class.

If you omit the **base** keyword, you are effectively referring to **this**, which could lead to a stack overflow as your method repeatedly calls itself.

Abstract Classes

- The **abstract** modifier indicates that an item has missing or incomplete implementation
- Use the **abstract** modifier in a **class** declaration to indicate that a class is intended to be used *only as a base class* for other classes
- **Abstract** classes can't be instantiated
- *Abstract members* within an *abstract* class must be implemented by non-abstract derived classes
- Derived classes receive:
 - Zero or more *concrete* methods/properties that they inherit
 - Zero or more *abstract* methods/properties that they inherit and must implement if they are a non-abstract class



QA

Abstract classes example – Part 1

An abstract class can contain abstract members:

```
public abstract class Shape
{
    // concrete properties and methods

    0 references
    public abstract void Draw();
    // abstract methods have no body
    // They must be overridden and implemented
    // in a non-abstract derived class
}
```

The abstract member must be implemented in a non-abstract derived class:

```
public class Rectangle : Shape
{
}
```

 CS0534 'Rectangle' does not implement inherited abstract member 'Shape.Draw()'

QA

Good practice in OO design is to factor out as much common data and behaviour as possible into a shared base class. If this base class becomes so general or abstract that it is used only as a framework by derived classes and is never instantiated, that class is known as an abstract class.

For example, the author of the Shape class has no real idea of how a specific shape will be drawn, nor can they possibly know how to calculate the area of a specific shape. However, they can determine that all shapes can be drawn and have their area calculated. Therefore, they can add an abstract definition of this behaviour without actually providing any implementation at all.

Derived classes must then replace (via overriding) the abstract member with a concrete implementation.

An abstract class can contain anything a non-abstract class can contain, such as instance variables and instance methods. Abstract classes can also contain abstract members. Abstract members do not contain any implementation code.

Abstract classes example – Part 2

Use the override keyword to implement the member:

```
public class Rectangle : Shape
{
    public override void Draw()
    {
        // implementation code goes here
    }
}
```

QA

Good practice in OO design is to factor out as much common data and behaviour as possible into a shared base class. If this base class becomes so general or abstract that it is used only as a framework by derived classes and is never instantiated, that class is known as an abstract class.

For example, the author of the Shape class has no real idea of how a specific shape will be drawn, nor can they possibly know how to calculate the area of a specific shape. However, they can determine that all shapes can be drawn and have their area calculated. Therefore, they can add an abstract definition of this behaviour without actually providing any implementation at all.

Derived classes must then replace (via overriding) the abstract member with a concrete implementation.

An abstract class can contain anything a non-abstract class can contain, such as instance variables and instance methods. Abstract classes can also contain abstract members. Abstract members do not contain any implementation code.

Abstract members

- Abstract *members* are declared using the **abstract** modifier and a *signature only*
- They do not contain any implementation code
- A class with even a single abstract member must be declared as abstract and cannot be instantiated
- Each derived class provides its own implementation for the abstract member or declares the inherited member as abstract and itself as an abstract class

```
public abstract class Shape
{
    // concrete properties and methods
    0 references
    public Color Colour { get; set; }

    // abstract method
    1 reference
    public abstract void Draw();

    // abstract property
    0 references
    public abstract double Area { get; }

    // abstract members have no body
    // They must be overriden and implemented
    // in a non-abstract derived class
}
```

QA

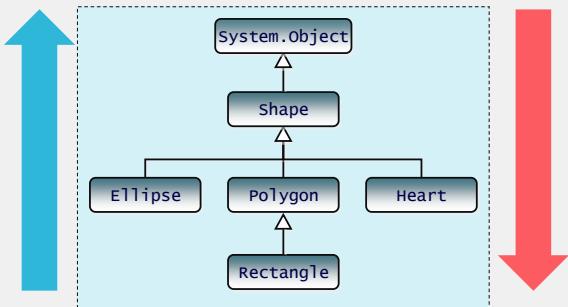
An abstract method is a method that cannot meaningfully be implemented by a class. Any class that contains one or more abstract methods is an abstract class and must be marked with the **abstract** keyword. A concrete derived class must implement all of the abstract methods of its base class. The declaration of an abstract method in an abstract base class forces all concrete descendants of that class to implement that method. Any derived class that does not implement all of the abstract methods of a base class is implicitly abstract and cannot be instantiated.

An abstract method has no method body; just a signature that includes the keyword **abstract**.

If ClassA is abstract and ClassB derives from ClassA and overrides all of ClassA's abstract members (making ClassB a concrete class), then any class that derives from ClassB does not need to provide its own implementation for ClassA's abstract methods. This is because ClassB has overridden all of the necessary members. Any further deriving classes can override these members if they require.

Casting derived and base classes

- An object of a **derived** class can be treated as an object of a **base** class without explicit casting. This is known as an **up-cast** and is safe
- An object of a **base** type needs to be explicitly cast to be used as a **derived** type. This is known as a **down-cast** and is potentially unsafe



QA

Every Rectangle is a Polygon and every Polygon is a Shape. It is therefore safe to use a variable declared as a Shape and pass a Polygon or Rectangle instance. This is an up-cast and happens implicitly.

The reverse is not true. Not every Shape is a Polygon. A Shape may be an Ellipse or a Heart shape. You must therefore perform a down-cast explicitly. If the type is incompatible, an exception will be thrown.

UP-Casting and Down-Casting

```
Drawing app = new Drawing();
Ellipse e = new Ellipse();
Rectangle r = new Rectangle();
Heart h = new Heart();

app.Shapes.Add(e); // implicit upcasting
app.Shapes.Add(r); // implicit upcasting
app.Shapes.Add(h); // implicit upcasting

Shape s = app.Shapes[0];
s.Draw(); // an Ellipse is drawn

Ellipse ell = (Ellipse)s; // explicit downcasting
double circumference = ell.Circumference;

Console.WriteLine(circumference);
```



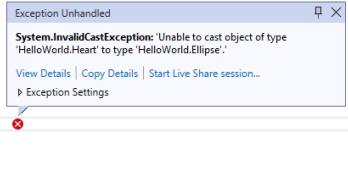
```
Drawing app = new Drawing();
Ellipse e = new Ellipse();
Rectangle r = new Rectangle();
Heart h = new Heart();

app.Shapes.Add(e); // implicit upcasting
app.Shapes.Add(r); // implicit upcasting
app.Shapes.Add(h); // implicit upcasting

Shape s = app.Shapes[2];
s.Draw(); // a Heart is drawn

Ellipse ell = (Ellipse)s; // explicit downcasting
double circumference = ell.Circumference;

Console.WriteLine(circumference);
```



QA

You can only invoke methods and properties of the base type when working with a base type reference. For example, in the code above the compiler will check to make sure that the method `Draw` is defined in the base **Shape** class. But what happens when we want to call a method or property that is defined in the derived class, such as the `Circumference` property defined in `Ellipse`?

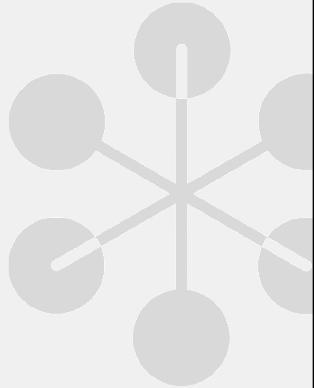
In this case, a down-cast must be performed explicitly. This is potentially unsafe. What happens if the `Shape` extracted from the collection is not an `Ellipse`? The cast to `Ellipse` would fail.

To summarise, it is always possible to use a base reference to refer to a derived object (and to implicitly convert a derived reference to a base reference, which is known as an “up cast”). Conversions down the hierarchy (down-casts) require explicit casts which may throw an exception. Extra checking can be performed to prevent the exception being thrown.

Safe Downcasting

To prevent an **InvalidCastException** being thrown, you can use the following operators:

- is
- as



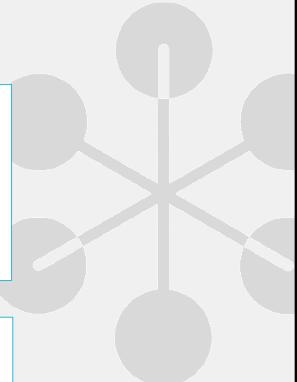
QA

The 'is' operator

The **is** operator checks if the result of an expression is compatible with a given type or matches a pattern:

```
Shape shape = app.Shapes[0];  
  
if (shape is Ellipse ellipse)// declaration pattern  
{  
    double circumference = ellipse.Circumference;  
    Console.WriteLine(circumference);  
}  
  
if (shape is not null)// type pattern null check  
{  
    Console.WriteLine(shape.Area);  
}
```

QA



The **is** operator checks if the run-time type of an expression result is compatible with a given type **T**. It returns true when an expression is not null and the run-time type matches the type **T** or is derived from type **T**. If an implicit reference conversion exists or a boxing or unboxing conversion exists, it also returns true.

From C# 7.0 onwards, you can also use the **is** operator to test an expression against a pattern. The example uses the *declaration pattern* whereby you declare a local variable to store the cast expression result if the type match succeeds.

From C# 9.0 onwards, you can use the **not**, **and**, and **or** pattern combinators to create *logical patterns*.

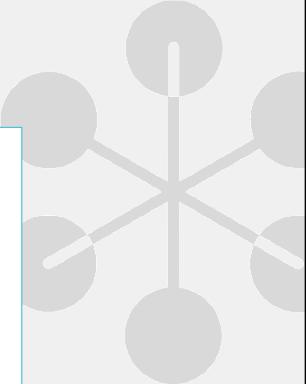
Best practice recommendation: use the pattern matching syntax whenever possible because it combines the test and the assignment in a single statement.

The 'as' Operator

- The **as** operator explicitly converts the result of an expression to a given type
- If the conversion isn't possible, the **as** operator returns **null**

```
Ellipse? ellipse = shape as Ellipse;  
if (ellipse != null)  
{  
    double circumference = ellipse.Circumference;  
    Console.WriteLine(circumference);  
}  
  
if (ellipse is not null)  
{  
    double circumference = ellipse.Circumference;  
    Console.WriteLine(circumference);  
}
```

QA



The **as** operator prevents an exception being thrown. It returns the cast expression result or null if the conversion is not possible.

The **!=** check uses any overloaded equality operators which may have been customized to return unexpected results when comparing to null.

The **is not null** check uses reference equality and will always return the expected outcome of a null test.

The Object Class

The ultimate base class of all .NET classes is **System.Object**.

A class implicitly inherits from **Object** if no base class is explicitly specified.

Object contains *virtual* methods that are commonly *overridden* in derived classes:

- **Equals**: Supports object comparisons
- **Finalize**: Performs clean-up before garbage collection
- **GetHashCode**: Generates a number to support the use of a hash table
- **ToString**: Provides a human-readable text string



QA

Overriding Object methods example

```
public class Book
{
    5 references
    public string Title { get; set; }

    5 references
    public string Author { get; set; }

    3 references
    public long ISBN { get; set; }

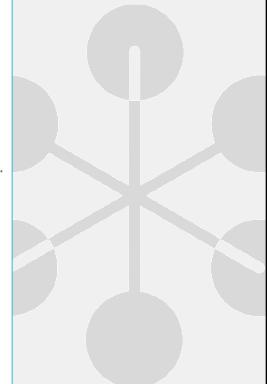
    1 reference
    public override bool Equals(object? obj)
    {
        // If this and obj do not refer to the same type, then they are not equal.
        if (obj.GetType() != this.GetType()) return false;

        // Return true if Title and Author fields match.
        var other = (Book)obj;
        return (this.Title == other.Title) && (this.Author == other.Author);
    }

    1 reference
    public override int GetHashCode()
    {
        string hashString = ISBN.ToString()[..9];// get first 9 digits
        return int.Parse(hashString);
    }

    1 reference
    public override string ToString()
    {
        return $"Title={Title}, Author={Author}";
    }
}
```

QA



The **Book** class implicitly inherits from **System.Object**

The *virtual Equals, GetHashCode* and **ToString** methods inherited from **Object** have all been *overridden*.

Overriding Object methods example

```
Book b1 = new();
b1.Author = "J K Rowling";
b1.Title = "Harry Potter and the Philosopher's Stone";
b1.ISBN = 9780590353403;

Book b2 = new();
b2.Author = "J K Rowling";
b2.Title = "Harry Potter and the Philosopher's Stone";
b2.ISBN = 9780590353403;

Console.WriteLine(b1.ToString()); // Title=Harry Potter and the Philosopher's Stone, Author=J K Rowling
Console.WriteLine(b1.GetHashCode()); // 9780590353403
Console.WriteLine(b1.Equals(b2)); // True (value equality)
Console.WriteLine(b1 == b2); // False (reference equality)

Book b3 = b1;
Console.WriteLine(b1.Equals(b3)); // True (value equality)
Console.WriteLine(b1 == b3); // True (reference equality)
```

QA

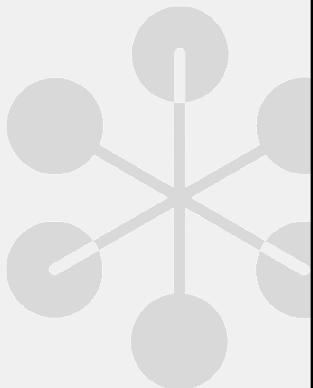
The overridden **Equals** method compares the Author and Title properties. If they match, it deems the two book objects to be equal.

The **==** operator compares the object references. Book **b1** and Book **b2** are different objects in memory and therefore this evaluates as *False*.

Book **b1** and Book **b3** point to the same object in memory and therefore this evaluates as *True*.

Sealed classes and members

- All classes can be inherited from unless the **sealed** modifier is applied
- All virtual members can be overridden anywhere within the inheritance hierarchy unless the sealed modifier is applied
- Structs are implicitly sealed and so cannot be inherited



QA

Sealed Classes

ClassA is not sealed so can be used as a base class:

```
1 reference
public class ClassA
{
}
```

ClassB inherits from ClassA but is marked as **sealed**:

```
1 reference
public sealed class ClassB : ClassA
{
}
```

No classes are allowed to inherit from a sealed class:

```
0 references
public class ClassC : ClassB
{
}
```

CS0509: 'ClassC': cannot derive from sealed type 'ClassB'
Show potential fixes (Alt+Enter or Ctrl+.)

QA

Note: You cannot seal an abstract class.

Sealed methods

```
1 reference
public class ClassX
{
    2 references
    protected virtual void F1() { Console.WriteLine("X.F1"); }
    2 references
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

1 reference
class ClassY : ClassX
{
    1 reference
    sealed protected override void F1() { Console.WriteLine("Y.F1"); }
    2 references
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

0 references
class ClassZ : ClassY
{
    // Attempting to override F1 causes compiler error CS0239.
    2 references
    protected override void F1() { Console.WriteLine("Z.F1"); }

    // Overriding F2 is allowed. ✘ CS0239 'ClassZ.F1()': cannot override inherited member 'ClassY.F1()' because it is sealed
    2 references
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

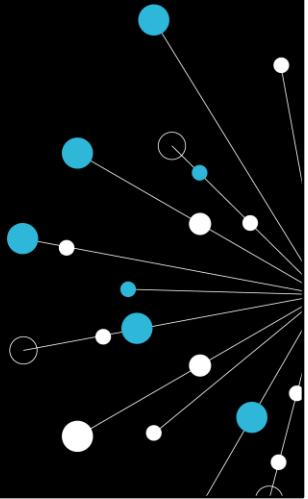
QA

Once a virtual member is sealed, it cannot be further overridden. You must use the sealed modifier with the override modifier for members.

Summary

- Inheritance
- Derived constructors
- Polymorphism
- Virtual members and overriding
- Invoking base class functionality
- Abstract classes
- Casting types: Up-casting, down-casting, is and as operators
- Overriding System.Object methods
- Sealed classes and members

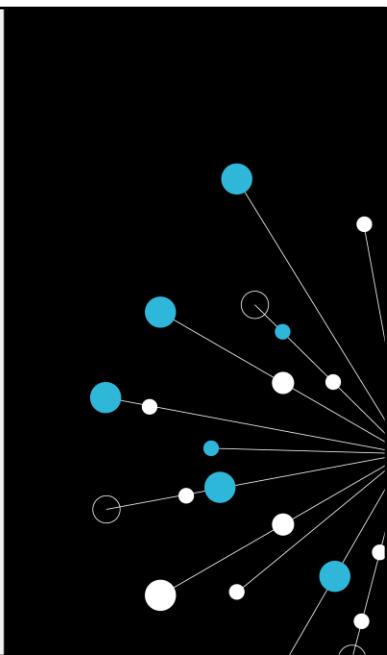
QA



Hands on Lab

Exercise 03 Inheritance and Abstract Classes

QA



04 Interfaces

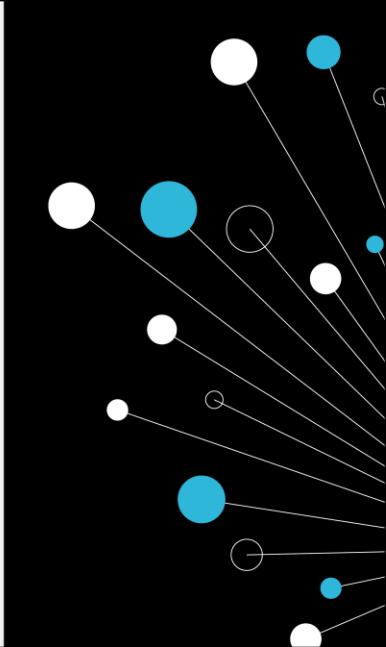
C# for HAAS F1



Learning objectives

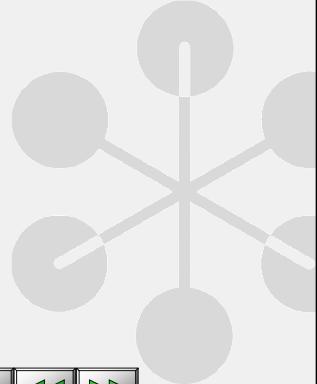
- Interfaces
- Implementing interfaces
- Polymorphism
- Multiple interfaces
- Default implementation
- Interface inheritance
- Member collisions
- Explicit implementation

QA



Interfaces

- An interface contains definitions for a group of related functionalities
- A class or struct that implements the interface must provide the implementation code for all members
- From C# 8.0, an interface member can provide a default implementation for a member
- A class or struct can implement many interfaces, whereas a class can only inherit from one base class and structs cannot inherit from any base classes
- Interfaces conventionally are identified with a capital 'I', followed by a verb that describes the group of functionalities e.g., IComparable, IControllable, IDisposable,, and IPlayable



QA

Classes in different inheritance hierarchies can implement the same interface to create a common group of functionalities across different classes.

Interfaces are implemented. They are not instantiated.

Interfaces cannot have any instance fields.

Defining an interface

- Define an interface using the **interface** keyword
- Interfaces can contain *methods, properties, indexers, and events*
- Interface members are implicitly **public** and **abstract**

```
public interface IDrawable
{
    void Draw(Graphics g); // no implementation code
}
```



QA

Implementing an interface

- List the interfaces after a colon and the base class (if also using inheritance)
- All non-default members must be implemented

```
public abstract class Shape
{
    1 reference
    public abstract double Area { get; }
}
```

```
public interface IDrawable
{
    1 reference
    void Draw(Graphics g); // no implementation code
}
```

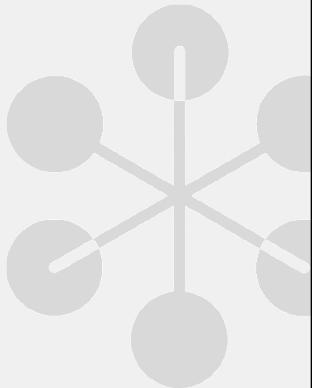
```
public class Rectangle : Shape, IDrawable
{
    1 reference
    public int Width { get; set; }
    1 reference
    public int Height { get; set; }
    1 reference
    public override double Area => Width * Height;

    1 reference
    public void Draw(Graphics g)
    {
        // implementation code goes here;
    }
}
```

QA

Polymorphism

- An interface defines a collection of related functionalities
- A class or struct that implements an interface 'can do' those functions, i.e., they play that role
- An interface type can be used as a method parameter, return type, or as the type in a generic collection
- Any implementing class or struct can be used where the interface type is expected



QA

Polymorphism example

```
Graphics canvas = new();

void ProcessDrawable(IDrawable id)
{
    if (id is not null)
    {
        id.Draw(canvas);
    }
}

List<Shape> shapes = new() { new Rectangle(), new Rectangle() };
foreach (Shape shape in shapes)
{
    if (shape is IDrawable s)
    {
        s.Draw(canvas);
        //or:
        ProcessDrawable(shape as IDrawable);
    }
}
```

QA

Multiple interfaces

A class or struct can implement multiple interfaces

```
public interface IComparable<T> {
    int CompareTo(T obj);
}
```

```
public interface IDrawable
{
    void Draw(Graphics g); // no implementation code
}
```

```
public class Rectangle : Shape, IDrawable, IComparable<Rectangle>
{
    3 references
    public int Width { get; set; }
    1 reference
    public int Height { get; set; }
    1 reference
    public override double Area => Width * Height;

    0 references
    public int CompareTo(Rectangle? other)
    {
        return Width - other.Width;
    }
    3 references
    public void Draw(Graphics g)
    {
        // implementation code goes here;
    }
}
```

QA

Default implementation

- From C# 8.0, interface members can provide a default implementation
- IEmployee** provides *default implementation* for the **GetTaxAmount** method

```
IEmployee employeeA = new Employee(2500);
Console.WriteLine($"The Tax amount is {employeeA.GetTaxAmount()}");
Console.ReadKey();

Employee employeeB = new Employee(2500);
Console.WriteLine($"The Tax amount is {employeeB.GetTaxAmount()}");
Console.ReadKey();
```

CS1061 'Employee' does not contain a definition for 'GetTaxAmount' and no accessible extension method 'GetTaxAmount' accepting a first argument of type 'Employee' could be found (are you missing a using directive or an assembly reference?)

```
public interface IEmployee
{
    1 reference
    public int ID
    {
        get;
        set;
    }
    2 references
    public string Name
    {
        get;
        set;
    }
    3 references
    public double Salary
    {
        get;
        set;
    }
    0 references
    public double GetTaxAmount()
    {
        return Salary * 0.05;
    }
}
```

```
public class Employee : IEmployee
{
    0 references
    public Employee(double salary)
    {
        Salary = salary;
    }
    1 reference
    public int ID { get; set; }
    1 reference
    public string Name { get; set; }
    3 references
    public double Salary { get; set; }
}
```

QA

The **GetTaxAmount** method does not have to be implemented in the implementing class (**Employee**) because the interface provides a default implementation.

This feature enables existing interfaces to be extended without breaking existing implementing classes and structs.

To access the default implementation, the declared type must be the **interface** type not the implementing class or struct type.

Interfaces can inherit from interfaces

- **IFullySteerable** inherits **ISteerable**
- Any implementing class or struct must implement all the members of both **ISteerable** and **IFullySteerable**

```
public interface ISteerable
{
    1 reference
    void TurnLeft();
    1 reference
    void TurnRight();
}
```

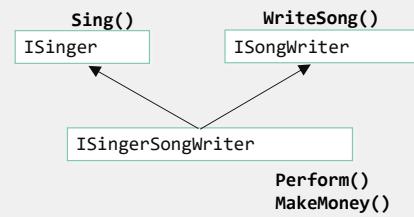
```
public interface IFullySteerable : ISteerable
{
    1 reference
    void GoUp();
    1 reference
    void GoDown();
}
```

```
public class Drone : IFullySteerable
{
    1 reference
    public void GoDown()
    {
        // go down implementation;
    }
    1 reference
    public void GoUp()
    {
        // go up implementation;
    }
    1 reference
    public void TurnLeft()
    {
        // turn left implementation;
    }
    1 reference
    public void TurnRight()
    {
        // turn right implementation;
    }
}
```

QA

Multiple Interface inheritance

- **ISingerSongWriter** inherits both **ISinger** and **ISongWriter**
- Implementing classes and structs must implement:
 - Sing
 - WriteSong
 - Perform
 - MakeMoney



QA

Multiple interface member collisions

- Member collisions can arise when multiple interfaces use the same member name for semantically different functionality
- You can only implement one version of the member using *implicit* interface implementation

```
public interface ICowboy
{
    0 references
    void Draw(Graphics g);
}
```

```
public interface IDrawable
{
    4 references
    void Draw(Graphics g);
}
```

```
public class CowboyShape : ICowboy, IDrawable
{
    4 references
    public void Draw(Graphics g)
    {
        // only one implementation;
    }
}
```

QA

Explicit Interface Implementation

You can implement interface members *explicitly*, which includes the interface name as part of the member name

```
public class CowboyShape : ICowboy, IDrawable
{
    // references
    public void Draw(Graphics g)
    {
        // implicit drawable implementation;
        Console.WriteLine("Drawing a cowboy shape");
    }
    // reference
    void ICowboy.Draw(Graphics g)
    {
        // explicit cowboy implementation;
        Console.WriteLine("Reach for the sky, mister!");
    }
}
```

```
CowboyShape cs = new CowboyShape();
cs.Draw(canvas); // Drawing a cowboy shape

IDrawable id = cs;
id.Draw(canvas); // Drawing a cowboy shape

ICowboy ic = cs;
ic.Draw(canvas); // Reach for the sky, mister!
```

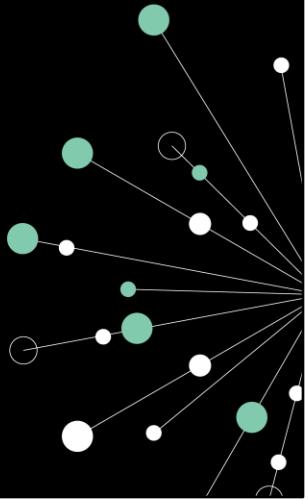
QA

You do not include an access modifier on an explicitly implemented member. The member is implicitly public through the interface reference.

Summary

- Interfaces
- Implementing interfaces
- Polymorphism
- Multiple interfaces
- Default implementation
- Interface inheritance
- Member collisions
- Explicit implementation

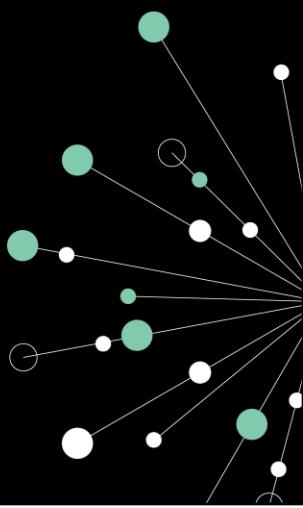
QA



Hands on Lab

Exercise 04 Interfaces

QA



05 SOLID Principles

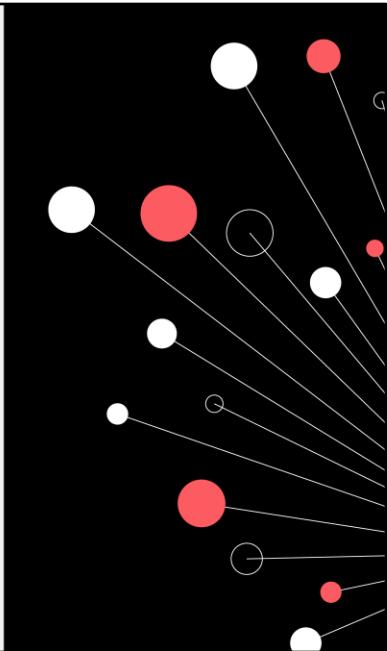
C# for HAAS F1



Learning objectives

- Be reminded of programming best practice
- To understand the main principles in refactoring classes
- Leading into Design Patterns which embody those principles

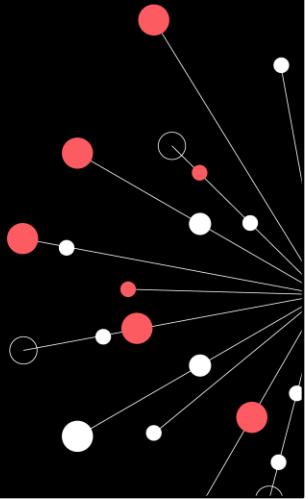
QA



Session Content

- C# Best Practice
- Experienced Developer best Practice
- Symptoms of a degrading design
- Background to SOLID Principles
 - Single-Responsibility Principle
 - Open/Closed Principle
 - Liskov Substitution Principle
 - Interface Segregation Principle
 - Dependency Inversion Principle

QA



C# Best Practice Top 10! - Part 1

C# is a versatile and powerful programming language, and following best practices can significantly enhance code quality, maintainability, and readability. Here are the top 10 C# coding best practices:

1. **Follow Naming Conventions**
2. **Keep Methods Short and Focused**
3. **Use Properties Instead of Public Variables**
4. **Utilize Exception Handling Properly**
5. **Apply the Principle of Least Privilege**



QA

1. Follow Naming Conventions:

- Use **PascalCase** for class names and method names.
- Use **camelCase** for local variables and method arguments.
- Use **UPPER_CASE** for constants.
- Names should be descriptive and clear about their purpose without being overly verbose.

2. Keep Methods Short and Focused:

- Each method should perform a single function. This makes methods easier to test, debug, and understand.

3. Use Properties Instead of Public Variables:

- Prefer properties with getters and setters to access fields of a class since properties provide better control over how values are set and retrieved.

4. Utilize Exception Handling Properly:

- Use try-catch blocks judiciously and avoid using exceptions for control flow.
- Always catch specific exceptions rather than general exceptions where possible.

5. Apply the Principle of Least Privilege:

- Limit access levels by making class members private or protected unless they genuinely need to be public.

C# Best Practice Top 10! - Part 2

6. Avoid Magic Numbers and Strings
7. Document Your Code
8. Use Asynchronous Programming Wisely*
9. Practice Immutable Object Patterns When Appropriate
10. Adhere to SOLID Principles...**



QA

6. Avoid Magic Numbers and Strings:

- Use named constants instead of hard-coding numbers or strings that appear in multiple places. This makes the code easier to maintain and understand.

7. Document Your Code:

- Use XML comments for documentation to provide summaries for classes, methods, and parameters. This documentation integrates with IDEs like Visual Studio and helps other developers understand your code quickly.

8. Use Asynchronous Programming Wisely*:

- Utilize async and await for asynchronous programming to keep UI responsive and improve performance for I/O-bound tasks.

9. Practice Immutable Object Patterns When Appropriate:

- When an object does not need to change after it is created, making it immutable can reduce bugs and design complexity.

10. Adhere to SOLID Principles...**

* There's a session for this topic coming soon

**A detailed discussion based on SOLID Principles is coming up soon!

Experienced C# Devs Best Practice Top 10! -Part 1

For experienced C# developers looking to refine their craft further, best practices should emphasize advanced programming techniques, architectural design, and efficient resource management. Here are the top 10 best practices tailored for experienced C# developers:

1. **Leverage Advanced Language Features Wisely**
 - Like LINQ, extension methods, and expression-bodied member
2. **Optimize Asynchronous and Parallel Programming**
 - Utilize Task.WhenAll, Task.WhenAny, Parallel.For, and Parallel.ForEach to maximize performance
3. **Deep Dive into Memory Management**
 - Apply best practices in memory management, particularly with regard to IDisposable
4. **Implement Design Patterns and Principles Thoughtfully***
 - Apply design patterns where they are genuinely beneficial

QA



- There's a session for this topic coming soon

1. Leverage Advanced Language Features Wisely:

Use features like LINQ, extension methods, and expression-bodied members to write concise and readable code. However, ensure that their use improves clarity and performance where applicable.

2. Optimize Asynchronous and Parallel Programming*:

Go beyond basic async and await usage. Utilize Task.WhenAll, Task.WhenAny, Parallel.For, and Parallel.ForEach to maximize performance in multi-threaded and asynchronous scenarios, while ensuring proper exception handling and synchronization.

3. Deep Dive into Memory Management:

Understand and apply best practices in memory management, particularly with regard to IDisposable, finalizers, and the effective use of using statements to manage resources efficiently.

4. Implement Design Patterns and Principles Thoughtfully*:

Apply design patterns where they are genuinely beneficial, rather than fitting problems to patterns. Choose patterns that appropriately address your architectural needs and understand their trade-offs.

Experienced C# Devs Best Practice Top 10! - Part 2

5. **Use Dependency Injection Effectively***
 - Utilize Dependency Injection (DI) to decouple class dependencies
6. **Emphasize on Code Testability and Robust Unit Tests**
 - Write testable code and build comprehensive unit tests
7. **Adopt Advanced Architectural Styles**
 - To address complex business needs effectively.
8. **Refactor Legacy Code With Strategy**
 - Develop a strategy for refactoring legacy systems. Introduce improvements incrementally and safely.
9. **Integrate Secure Coding Practices**
 - Prioritize security in the development lifecycle.
10. **Stay Updated and Contribute to the Community**
 - Keep abreast of the latest C# features and .NET technologies. Engage with the community through forums, blogs, and conferences.



QA

* There's a session for this topic coming soon

5. **Use Dependency Injection Effectively:**

Utilize Dependency Injection (DI) to decouple class dependencies, facilitate easier unit testing, and manage class lifecycles, particularly in large applications or those built on frameworks like ASP.NET Core.
6. **Emphasize on Code Testability and Robust Unit Tests:**

Write testable code and build comprehensive unit tests using frameworks like NUnit or xUnit. Mock dependencies using tools like Moq or NSubstitute. Ensure that tests cover edge cases and failure modes, not just the happy paths.
7. **Adopt Advanced Architectural Styles:**

Depending on project requirements, explore and adopt architectural styles such as Clean Architecture, CQRS (Command Query Responsibility Segregation), or Event Sourcing to address complex business needs effectively.
8. **Refactor Legacy Code With Strategy:**

Develop a strategy for refactoring legacy systems. Introduce improvements incrementally and safely by employing techniques like strangling patterns, feature toggles, or branch by abstraction.
9. **Integrate Secure Coding Practices*:**

Prioritize security in the development lifecycle. Apply secure coding practices, such as validating input, using secure communication protocols, and managing data securely to prevent vulnerabilities like SQL injection, cross-site scripting, or data breaches.
10. **Stay Updated and Contribute to the Community:**

Keep abreast of the latest C# features and .NET technologies. Engage with the community through forums, blogs, and conferences. Share knowledge, contribute to open-source projects, and learn from peer reviews to continuously refine your skills.

Introduction to software architecture and dependencies

System architectures are multi-levelled

What causes an application to go 'bad' and become less stable?

At high level – feel the overall shape and structure

- Still a language independent view

At lower level – actual modules of code with interdependencies

- We can see design patterns, classes and components (our domain)

Original solution elegantly designed and implemented. However:

- Initial bug fixes/enhancements (hacks) keep a veneer of elegance
- Over time codebase degrades, ugly features appear
- More time and effort needed for even simple changes
- New re-design needed but moving goalposts, new design has V1.0 flaws

QA

In this chapter we are going to consider software architecture at a few levels before deciding where as developer/implementers we can best affect the robustness of the code generated.

Architecture is multilevelled. At high levels there are the patterns that define the overall shape of an application and the structure related to the purpose of the application. At lower levels live the actual modules of compiled code with their interdependencies. This is where we can look at the nitty-gritty of design patterns, components and classes. We should recognise this is a huge area and we can provide a little knowledge and understanding, but often a little knowledge is dangerous. Much further reading is required, thousands of documents on the internet expand on the principles referred to in this chapter.

What is it that causes software to go awry, to become less stable? An original design may have been very elegant and clean and with no late design amendments to fit requirement changes. It was developed and made its way to a first release as V1.0. But at some variable speed the software slowly goes 'bad'. Initially a few hacked bug fixes and quick enhancements do not spoil the elegance of the original design. But over time it gets worse, possibly at an increasing rate until ugly features dominate the code. The code base becomes harder to maintain and before long the time/effort required to make the simplest of changes cause the owners and engineers to consider a complete redesign. This sort of redesign is often flawed as the designers are working against 'moving goalposts' as the original system is still evolving. There are then inherent problems latent in the new design before it even gets to its release date. When that happens – often after delays – the inherent problems surfacing in the new design mean people are already considering the next redesign/rebuild.

Symptoms of the design degrading

Extensive literature (on the web) on this subject

Four (partly related) symptoms are a clue that design has degraded

- Rigidity – software difficult to change even in a simple way
- Fragility (related) – tendency to break in many places when updated
- Immobility – inability to reuse software components
- Viscosity

Multiple ways of making a change

Easier to 'hack' a change than preserve design ('high' viscosity)

Environment features can push engineers to non-optimal solution



QA

The writers on this subject suggest four main symptoms that tell you when a design has gone bad. They are related to each other in a way that will become clearer through the chapter.

The symptoms are known as rigidity, fragility, immobility and viscosity.

These phrases were coined by a pioneering writer on OO Design, Robert C Martin in 2000 and now in general use.

Rigidity & Fragility

Rigidity

- Problematic to update in simple way
- Changes cascade to related changes in dependent routines
- Time and effort needed to follow a chain of repercussions
- Difficulty predicting duration of planned changes
- Owners become fearful of allowing non-critical changes
- Official rigidity sets in

Fragility

- Tendency to break in multiple places
- Failure can occur in areas with no conceptual relationship
- Fear increases of software always failing after an update
- Probability of further breakdowns increase

QA

This is the tendency for software to be problematic to update even in a simple way. Changes seem to cause a cascade of little related changes in dependent routines. What starts out looking like a tiny amendment becomes a multi-day change routine affecting many modules of code as developers follow the repercussions through the application.

When this behaviour is exhibited, decision makers become fearful of allowing anything of a non-critical nature to be fixed.

This is driven from the fact that they find it harder to predict, how long a change will take.

The software design begins to become a big drain on developer resources. If these concerns become so acute that they refuse to allow changes to software, then official rigidity has set in.

Basically, a design deficiency has ended up being the cause of negative management policy.

Rigidity is related to fragility.

This is a tendency of the software to break in multiple locations each time it is updated. Failures can occur in areas that have no conceptual relationship with the area that was changed.

This sort of error fills the minds of decision makers of managers with dread. Every time an update is authorised they live fear that the software will break in some unexpected way.

As the fragility increase, the probability of further breakages increase over time. This software becomes almost impossible to maintain.

Nearly every fix makes it worse, introducing more problems than are solved. Such software causes customers and owners to suspect that engineers have lost control of their software. Distrust abounds, and credibility is lost.

Immobility & Viscosity

Immobility

Inability to reuse components from other projects/parts of this one

A similar module needed to another one already written

It comes with too much baggage

Work involved and risks in separating needed from unneeded

Leads to rewrite and not reuse

Viscosity (aiming for 'low' viscosity)

Design viscosity

- Multiple ways of making a change work
- One way preserves the design the other (easier) does not
- So easier to do the wrong thing and harder to do the right thing

Environment viscosity

- Long re-build times, check-in check-out difficulties

QA

Immobility is the inability to reuse software from other projects or from parts of the same project. An engineer will decide that he needs a module that is similar to one that another engineer wrote. However, it also often happens that the module in question has too much baggage that it depends upon.

After investigation the engineers discover that the work involved and risk taken to separate the wanted parts of the software from the unwanted parts is too great and so the software is simply rewritten instead of reused.

Viscosity comes in two forms: design viscosity and environment viscosity.

When faced with a change, there is often more than one way to effect the change. One way might preserve the design, another does not – it is a ‘hack’. When the design preserving method is harder to employ than the ‘hack’, then the design is said to have ‘high viscosity’. It becomes perhaps easier to do the wrong thing, but harder to do the right thing.

Viscosity of environment comes about when the development environment is inefficient.

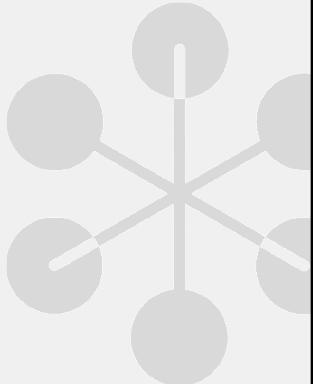
If compile times are long, engineers will be tempted to make changes that don’t force large recompiles, even though those changes are non-optimal from a design standpoint. If the source code control system takes ages to check-in just a few files, then engineers will be tempted to make the changes that require as few check-ins as possible, regardless of whether or not the original design is maintained.

These symptoms are the signs of poor architecture. Any application that exhibits them is suffering from a design that is breaking apart from the inside out. But what causes this to take place?

Changing requirements

What has caused these degradations of design?

- Usually, requirements changing in ways that were unanticipated
- Changes made urgently by engineers unfamiliar with original design
- Change successful but design ‘violated’
- Over time these accumulate



But software developers know that requirements will change

- So...
- Designs are partially at fault
- Need a way to make designs and code in them more resilient to change

QA

The immediate cause of the degradation of the design is usually recognised as a case of ‘requirements have been changing in ways that the initial design did not anticipate’. Often changes have to be made urgently, and perhaps by engineers who are not familiar with the original design philosophy. So, a change to the design might work, but ‘violates’ the original design.

Incrementally, as changes continue to be applied, these violations accumulate until the rot sets in.

But you cannot totally blame the drifting of the requirements for the degradation. As software engineers you know well enough that requirements inevitably change. Often the requirements document is one the most volatile documents in a project.

If designs are failing due to a constant demand of changing requirements, it is the designs that are (partially) at fault. A way is needed to make the designs more resilient to such changes and protect them from degrading.

Dependency management

What sort of changes cause design degradation?

- Usually changes that bring in new or unplanned ‘dependencies’
- All symptoms just covered are caused by improper dependencies between modules
- It is the dependency architecture that degrades
 - With it the ability to maintain the software

To delay degradation, dependencies must be managed

- ‘Dependency firewalls’ across which dependency does not propagate

OO has principles and techniques (design patterns) to:

- Build these firewalls
- Manage dependencies
- Maintain the dependency architecture

QA



What are the sorts of changes that cause designs to degrade? Usually changes that introduce new and/or unplanned for dependencies. All the symptoms just covered are either directly or indirectly caused by improper dependencies between the various modules of the software. It is actually the dependency architecture that is degrading, and with it the ability to easily maintain the software.

In order to delay any degradation of the dependency architecture, the dependencies between the modules must be managed. This management consists of the creation of ‘dependency firewalls’. Across such firewalls, dependencies do not propagate.

Object Oriented Design has many principles and techniques for building such firewalls, and for managing module dependencies. It is these principles and techniques that will be discussed in the slides that follow.

First we will examine the principles, and then the techniques (design patterns) that help maintain the dependency architecture of an application.

'SOLID' Principles

Five Dependency management principles for OO Programming/Design:

- Acronym 'SOLID' coined for these principles
- Applied together more likely to create a more maintainable extendable application
 - Guidelines applied to software to remove 'code smells'
 - Programmer refactors source code to be legible and extensible
- **S**ingle Responsibility principle (SRP)
- **O**pen/Closed principle (OCP)
- **L**iskov substitution principle (LSP)
- **I**nterface segregation principle (ISP)
- **D**ependency inversion principle (DIP)
- Part of overall strategy of Agile and Adaptive Programming



QA

The *SOLID principles* are five dependency management principles used in OO programming and design. The acronym was introduced from the work of Robert Martin in early 2000's. Each letter represents another three-letter acronym that describes one principle.

When working with software in which dependency management is handled badly, code can become rigid, fragile and difficult to reuse. Code which is difficult to modify, either to change existing functionality or add new features, susceptible to the introduction of bugs, particularly those that appear in a when another area of code is changed. If you follow the SOLID principles, you produce code that is more flexible and robust, with a higher possibility for reuse.

The following slides give an overview of the five principles.

Single Responsibility Principle (SRP)

- A 'responsibility' is a 'reason to change'
- Should be one reason only for class to change
 - Should have single purpose
 - All methods related to primary function
 - Find multiple responsibilities
 - Greater likelihood of change being needed
 - Split into new classes
- Represents a good way of identifying classes during the design phase
 - Reminds you to think of all the ways a class can evolve



QA

The Single Responsibility principle says that there should never be more than one reason for a class to change. Effectively this means you should design your classes so that each has a single purpose. It certainly does not mean that each class should have a single method but that all of the methods and properties in the class are directly related to the class's primary function. Where a class has multiple responsibilities, these should be separated into new classes.

When a class has multiple responsibilities, the likelihood that it will need to be changed increases. Each time a class gets modified the risk of introducing a bug grows. By concentrating on a single responsibility, the risk is limited.

If a class does two things well then when it is only needed for one of those purposes then it comes with baggage.

SRP is a simple and fairly intuitive principle, but in practice it is often hard to get right as examples that follow will show.

Consider a class that compiles and prints a report, it could be changed for two reasons. The content of the report might change, the format could. These two things are changing for different reasons; one is 'substantive', and one 'cosmetic'. SRP says that these two aspects of the class are really two separate responsibilities, and should therefore be in separate classes or modules. What you are trying to do is avoiding the coupling of two things that change for different reasons at different times.

If there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

SRP Example 1 – violation (three reasons for change)

```
public class CO2Meter {  
    public double co2Saturation { get; set; }  
  
    public void ReadCO2Level() {  
        using (MeterStream ms = new MeterStream("CO2")) {  
            int raw = ms.ReadByte();  
            co2Saturation = (double)raw / 255 * 100; // % calc  
        }  
    }  
  
    public bool CO2High() { return co2Saturation >= 2; }  
  
    public void ShowHighCO2Alert() {  
        Console.WriteLine(  
            $"CO2 high ({co2Saturation :F1}%)");  
    }  
}
```

QA

Talks to hardware device to monitor CO2 levels

1) CO2 monitoring hardware could change

2) Process could change to consider a temperature factor

3) Alerting improved beyond 'Console' output

The code above is a class that communicates with a hardware device to monitor the CO2 levels in some fluid. The class includes a method named "ReadCO2Level" that retrieves a value from a stream generated by the CO2 monitoring hardware. It converts the value to a percentage and stores it in the CO2Saturation property. The second method, "CO2High", checks the CO2 saturation to ensure that it does not exceed the maximum level of 2%. The "ShowHighCO2Alert" shows a warning that contains the current saturation value. There are at least three reasons for change within the CO2Meter class. If the CO2 monitoring hardware is replaced the ReadCO2Level method will need to be updated. If the process for determining high CO2 is changed, perhaps to include a temperature variable, the class will need updating. Finally, if the alerting requirements become more sophisticated than outputting text to the console, the ShowHighCO2Alert method will need to be rewritten.

Gather together the things that change for the same reasons. Separate those things that change for different reasons.

This is really just a way to define cohesion and coupling. You want to increase the cohesion between things that change for the same reasons, and to decrease the coupling between those things that change for different reasons.

SRP Example 1 – code refactored to three classes

```
public class CO2Meter {
    public double CO2Saturation { get; set; }
    public void ReadCO2Level() {
        using (MeterStream ms = new MeterStream("CO2")) {
            int raw = ms.ReadByte();
            CO2Saturation = (double)raw / 255 * 100; // % calc
        }
    }
} // end class
public class CO2SaturationChecker {
    public bool CO2High(CO2Meter meter) {
        return meter.CO2Saturation >= 2;
    }
} // end class
public class CO2Alerter {
    public void ShowHighCO2Alert(CO2Meter meter) {
        Console.WriteLine(
            $"CO2 high ({meter.CO2Saturation :F1}%)");
    }
} // end class
```

QA

CO2Meter injected

Refactored code has functionality split into three classes. The first is the CO2Meter class. This retains the CO2Saturation property and the ReadCO2Level method. You could split these into separate classes but we will keep them together as they are closely related. The other methods are removed so that the only reason for change is replacement of the monitoring hardware. The second class is named "CO2SaturationChecker". This class includes a single method that compares the CO2 level with a maximum acceptable value. The method is the same as the original except for the addition of a parameter that injects a CO2Meter object containing the saturation level to test. The only reason for the class to change is if the test process is changed. The final class is named "CO2Alerter". This displays an alert that includes the current oxygen saturation level. Again, a CO2Meter dependency is injected. The one reason for the class to change is if the alerting system is updated.

Note – this refactored code breaks other SOLID principles in order that the application of the SRP is visible. Further refactoring of this example is necessary to achieve compliance with the other four principles.

Application of the SRP changes your code considerably. Classes in your projects become smaller and cleaner. The number of classes present in a solution will increase accordingly, so it is important to organise them well using namespaces and project folders. The creation of classes that are tightly focused on a single purpose leads to code that is conceptually simpler to understand and maintain.

Another benefit of having small, cohesive classes is that the chances of a class containing bugs drops. This reduces the need for changes, so the code is less fragile. As the classes perform only one duty, multiple classes will work together to achieve larger tasks. Along with the other principles this permits looser coupling. It can also make it easier to modify the overall software, either by extending existing classes or introducing new, interchangeable versions.

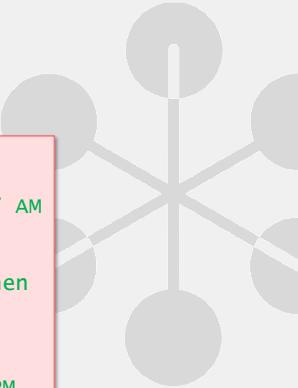
SRP EXAMPLE 2

Class has two responsibilities

- It is mixing 'opening and closing gate' functionality with providing core vehicle servicing role

```
public class PetrolStation {  
    public void OpenGate() {  
        // Open the gate if the time is later than 7 AM  
    }  
    public void Service(Vehicle vehicle) {  
        // Check if service station is opened and then  
        // complete the vehicle servicing  
    }  
    public void CloseGate() {  
        // Close the gate if the time has passed 7 PM  
    }  
}
```

QA



SRP Example 2 after refactoring

Class is mixing 'opening and closing barrier' functionality with providing core vehicle servicing task

QA

```
public interface IGateUtility {  
    void OpenGate();  
    void CloseGate();  
}  
public class PetrolStationUtility : IGateUtility {  
    public void OpenGate() { /* Open gate if after 7AM */ }  
    public void CloseGate() { /* Close gate at 7PM */ }  
}  
public class PetrolStation {  
    private IGateUtility gateUtil;  
    public PetrolStation(IGateUtility gateUtil) {  
        this.gateUtil = gateUtil;  
    }  
    public void OpenForService() { gateUtil.openGate(); }  
    public void DoService() {  
        // If service station is open then service vehicle  
    }  
    public void CloseForDay() { gateUtil.CloseGate(); }  
}
```

The refactored code above works as follows.

A new interface is defined that just contains 'gate' related utility methods.
A concrete class implements the interface and supplies Open/CloseGate functionality.

The PetrolStation when instantiated is told of an IGateUtility object, he stores its reference (the assignment in the constructor). When it needs utility object at start/end of day it invokes the Open Close functionality that has been moved to a different class PetrolStationUtility.

If the way gates get opened/closed needs to change, or there is a new fangled barrier introduced, the PetrolStation will not need updating, it can just be passed a new concrete implementation of IGateUtility by client code.

Open / Closed Principle (OCP)

OCP says make classes

- 'Open for extension'
 - New functionality added as new requirements generated
- 'Closed for modification'
 - Once developed, no modification except for bug fixes
- Appears contradictory...
- But achieved by referring to abstractions(interfaces typically) for dependencies rather than concrete classes
 - Interfaces fixed so classes depend on unchanging abstractions
 - Functionality added by new classes implementing the interfaces
- Applying OCP limits the need to change source code once tested
 - Reduces the risk of introducing new bugs to existing code



QA

The Open / Closed Principle (OCP) says that classes should be 'open for extension' but 'closed for modification'. 'Open to extension' means that you design your classes so that new functionality can be added as new requirements are generated. 'Closed for modification' basically means that once you have developed a class you should never modify it, except to correct bugs.

On initial viewing the two parts appear to contradict each other. However, if you correctly structure your classes and their dependencies you can add functionality without editing existing source code.

This is normally achieved by referring to abstractions (interfaces or abstract classes) for dependencies, rather than using concrete classes.

Such interfaces can be fixed once developed so the classes that depend upon them can rely upon unchanging abstractions. Functionality is then added by creating new classes that implement the interfaces.

Applying the key Open Closed principle to your applications limits the need to change source code once it has been written, tested and debugged. It reduces the risk of introducing new bugs to existing code, leading to more robust software. Another side effect of the use of interfaces for dependencies is reduced coupling and increased flexibility.

Open / Closed Principle (OCP) – violation

```
public class Logger {  
    public void Log(string message, string loggingType) {  
        switch (loggingType) {  
            case "Console":  
                Console.WriteLine(message);  
                break;  
            case "File":  
                // Code to send message to default printer  
                break;  
        }  
    }  
}
```

Could be a (changeable)
enum?

Switch needs updating if
new 'loggingType'

QA

The sample above is a basic module for logging messages. The Logger class has a single method that accepts a message to be logged and the type of logging to perform. The switch statement changes the action according to whether the program is outputting messages to the console or to the default printer. If you wished to add a third type of logging, perhaps sending the logged messages to a message queue or storing them in a database, you could not do so without modifying the existing code. Firstly, you would need to add new loggingType constants for the new methods of logging messages. Secondly, you would need to extend the switch statement to check for the new loggingTypes and output or store messages accordingly. This violates the OCP.

Open / Closed Principle (OCP) – code refactored

```
public class Logger {  
    private IMessageLogger messageLogger;  
    public Logger(IMessageLogger messageLogger) {  
        this.messageLogger = messageLogger;  
    }  
    public void Log(string message) {  
        messageLogger.Log(message);  
    }  
}  
public interface IMessageLogger { void Log(string message); }  
public class ConsoleLogger : IMessageLogger {  
    public void Log(string message) {  
        Console.WriteLine(message);  
    }  
}  
public class PrinterLogger : IMessageLogger {  
    public void Log(string message) {  
        // Code to send message to printer  
    }  
}
```

Dependency is only on type 'IMessageLogger'

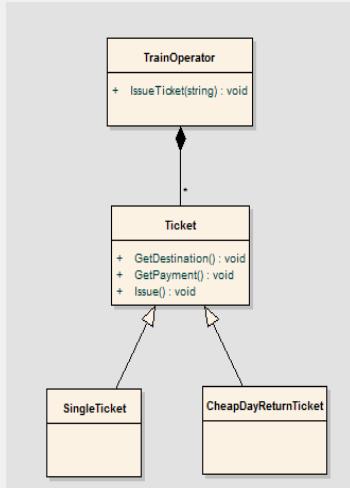
Concrete classes that implement the dependent interface

QA

We can easily refactor the logging code to achieve compliance with the OCP. Firstly, we need to remove any LoggingType enumeration, as this restricts the types of logging that can be included. Instead of passing the type to the Logger, we will create a new class for each type of message logger that we require. In the final code we will have two such classes, named "ConsoleLogger" and "PrinterLogger". Additional logging types could be added later without changing any existing code.

The Logger class still performs all logging but using one of the message logger classes described above to output a message. In order that the classes are not tightly coupled, each message logger type implements the IMessageLogger interface. The Logger class is never aware of the type of logging being used as its dependency is provided as an IMessageLogger instance using constructor injection.

OCP – Example 2 – separate the bits that change



```
class TrainOperator {
    public void IssueTicket
        (string type)
    {
        Ticket ticket = null;
        if (type == "Single")
            ticket= new SingleTicket();
        else if (type == "CheapDayRtn")
            ticket= new CheapDayRtnTicket(),
        ticket.GetDestination();
        ticket.GetPayment();
        ticket.Issue();
    }
}
```

This part changes with every new type of Ticket

A red oval highlights the conditional logic in the `IssueTicket` method, specifically the `if` and `else if` statements. A callout box below the code states: "This part changes with every new type of Ticket".

QA

The if / else construct will need updating for every new sort of ticket. This needs to be factored (separated) out.

OCP – Example 2 - Closed For Modification

```
class TrainOperator {
    public void IssueTicket
        (string type)
    {
        Ticket ticket = null;

        ticket =
            TicketFactory
                .CreateTicket(type);

        ticket.GetDestination();
        ticket.GetPayment();
        ticket.Issue();
    }
}
```

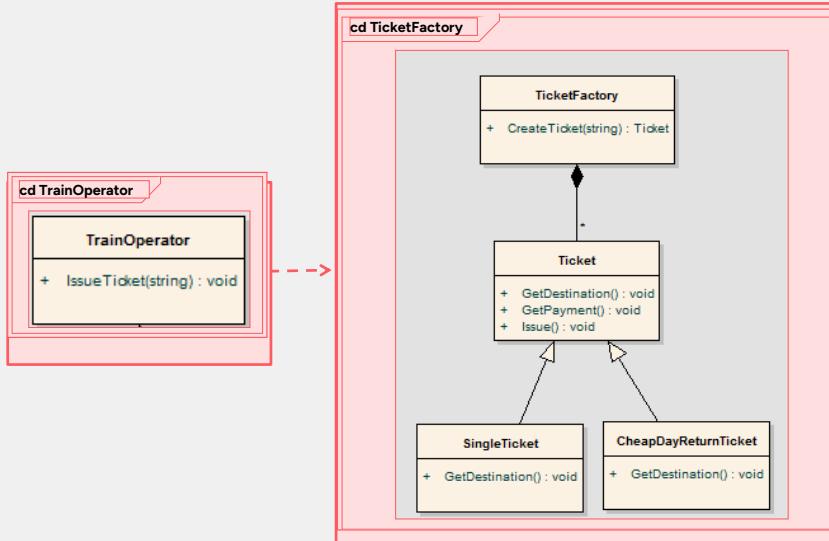
```
public static class TicketFactory {
    public Ticket CreateTicket
        (string type) {
        Ticket ticket = null;
        if (type == "Single")
            ticket
                = new SingleTicket();
        else if (type == "CheapDayRtn")
            ticket
                = new CheapDayRtnTicket();
        return ticket;
    }
}
```

QA

The CreateTicket() method is placed in a factory class method, the IssueTicket() method needs no more changes.

OCP – Example 2 - Open For Extension

QA



TrainOperator now depends on the TicketFactory class whose CreateTicket method returns a base class 'Ticket' type.

Liskov Substitution Principle (LSP)

LSP states:

- If program code is using a base class reference, then the reference to object can be replaced with a reference to a derived class object without affecting the functionality of the program code

Must ensure that any/all derived classes just extend without replacing functionality of base types

- Otherwise, the new classes can produce undesired effects when they are used in existing program code

Classic example of violation is surprisingly Rectangle and Square

- You would think a Square 'is a' Rectangle and is fully substitutable
- But not if Rectangle is mutable (see over)

LSP says:

- Will the client's perception that it is a Rectangle ever be broken if you pass in a Square object?



QA

The whole point of LSP is to be able to pass around a subclass as the parent class without any problems.

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.

The original statement by Barbara Liskov (1988) was:

"What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."

It was then paraphrased later as:

"Methods that use references to base classes must be able to use objects of derived classes without knowing it."

The importance of this principle becomes obvious when you consider the consequences of violating it.

If there is a method which does not conform to the LSP, then that method uses a reference to a base class, but must *know about all the derivatives* of that base class. Such a method violates the Open-Closed principle because it must be modified whenever a new derivative of the base class is created.

Also anytime you see code that takes in some sort of baseclass or interface and then performs a check such as "if (someObject is SomeType)", there's a very good chance that that's an LSP violation.

Liskov Substitution Principle (LSP) – class definitions

```
class Rectangle {  
    public int Width { get; private set; }  
    public int Height { get; private set; }  
    public virtual void SetWidth(int w) { Width = w; }  
    public virtual void SetHeight(int h) { Height = h; }  
    public int Area { get {return Width * Height;} }  
}  
class Square : Rectangle {  
    public override void SetWidth(int w) {  
        base.SetWidth(w);  
        base.SetHeight(w); // Squares have identical width/height  
    }  
    public override void SetHeight(int h) {  
        base.SetWidth(h); // Squares have identical width/height  
        base.SetHeight(h);  
    }  
}
```

First clue that a Square should perhaps not inherit from Rectangle might be the fact that a Square does not need both Height and Width member variables!

QA

SetWidth() of Rectangle must be virtual as the behaviour defined there is not suitable for a Square.

If you were able to do:

```
Square s = new Square();
```

s.setWidth(10) ...and it didn't change the Height as well it would not be a square anymore.

LSP is about following the contract of the base class.

Liskov Substitution Principle (LSP) – client code

What would the user (who doesn't know how the factory produces a 'Rectangle') expect this code to print?

- 9, 15 or 25?

```
public void PlaywithRectangle() {
    Rectangle r = RectangleFactory();
    // Client code not certain what sort of Rectangle it has
    // But should not need to know!
    r.setHeight(3);
    r.setWidth(5);
    console.WriteLine(r.Area);
}

private Rectangle RectangleFactory() {
    ....
    return new Square();
}
```

QA

The derived class has changed the behaviour of the base class – a clear violation.

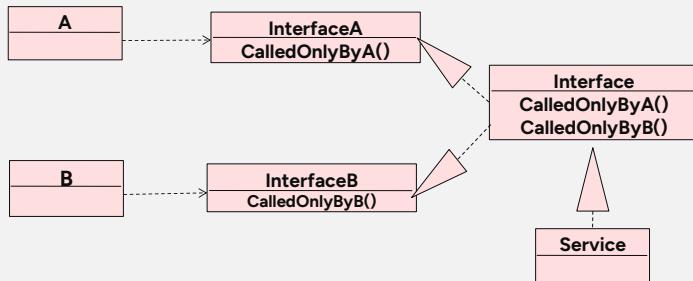
For instance, although the C# compiler allows it, in a sub class you shouldn't throw a new exception in an overridden method if the base class wasn't built to expect it. The sub class will have changed the behavior. Client code that is treating an instance of the sub class as a base class object may get an unanticipated exception thrown at it. The same goes for if the base class throws ArgumentNullException if an argument is missing and the sub class allows the argument to be null, also a LSP violation.



Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they do not use

- If they are, then changes in one part of the system can cause changes in completely unrelated areas



QA

A service that needs only one of the methods should not be dependent on an interface that demands two or more methods be implemented. The interface should be separated into two separate interfaces.

Interface Segregation Principle (ISP)

ISP splits large interfaces into smaller and more specific ones

Clients will only have to know of methods of interest to them

- These 'shrunken' interfaces often referred to as *role interfaces*

ISP helps to keep a system decoupled

- Easier to re-factor, change and redeploy



QA

The ISP principle was first used and formulated by Robert Martin in the 1990s while consulting at Xerox. Xerox had created a printer system that could perform a variety of tasks such as stapling and faxing. The software for this system was created from the ground up. As the software grew, making modifications became more and more difficult so that even the smallest change was incurring a significant redeployment time.

The design problem was that a single Job class was used by almost all of the tasks. Whenever a print job or a stapling job needed to be performed, a call was made to the Job class.

This Job class became very 'fat' with multitudes of methods specific to a variety of different clients. Because of this design, a staple job would know about all the methods of the print job, even though there was no use for them.

The solution Martin used to solve this problem utilised what is now known as the Interface Segregation Principle.

Applied to the Xerox problem, an interface layer between the Job class and its clients was added using the Dependency Inversion Principle (covered soon). Instead of having one large Job class, a Staple Job interface or a Print Job interface was created that would be used by the Staple or Print classes, respectively, calling methods of the Job class. Therefore, one interface was created for each job type, which were all implemented by the Job class.

Interface Segregation Principle (ISP) - violation

```
// interface segregation principle - bad example
interface IWorker {
    public void work();
    public void Eat();
}
class Worker : IWorker {
    public void work() { // ....working }
    public void Eat() { // ... eating in lunch break }
}
class SuperWorker : IWorker{
    public void work() { //.... working much more }
    public void Eat() { //.... eating in lunch break }
}
class Manager {
    IWorker worker;
    public void Setworker(IWorker w) { worker = w; }
    public void Manage() { worker.work(); }
}
```

What happens when a Robot (that Works but does not Eat) is introduced?

QA

When we design an application, we take care how we abstract a module which contains several submodules. Considering code implemented by a class, we can have an abstraction of the system done in an interface. But if we want to extend our application, adding another module that contains only some of the submodules of the original system, we are forced to implement the full interface (and to write some dummy methods). Such an interface is named fat or polluted. Having interface pollution is not a good solution and often produces inappropriate behavior in the system.

ISP says clients should not be forced to implement interfaces they don't use. Instead of one fat interface many smaller interfaces are preferred based on groups of methods, each one serving one submodule.

Study the example above of a violation of ISP. We have a Manager class which represents the person which manages the workers. Two types of workers: some average, some very efficient. Both types of workers work and they need a daily lunch break. But then some robots start to work for the company as well, but they don't eat so they don't need a lunch break. But the new Robot class needs to implement the IWorker interface because robots work. Unfortunately, if the Robot class implements IWorker then what do we about the Eat method?

This is why in this case the IWorker is considered 'polluted'.

If you keep the present design, the new Robot class is forced to implement the Eat method. We can write a dummy method which does nothing (maybe a 1 second lunch break per day), but this can have subtle undesired effects in the application, for example the reports will report more lunches taken than the number of people!

According to the ISP, a flexible design will not have polluted interfaces. In this example, the IWorker interface should be split in two different interfaces.

Interface Segregation Principle (ISP) – solution

```
// interface segregation principle - reworked code
interface Iworkable { public void work(); }
interface IFeedable { public void Eat(); }
interface Iworker : IFeedable, Iworkable { }
class Worker : Iworker {
    public void work() { // working }
    public void Eat() { // ... eating in lunch break }
}
class Robot : Iworkable {
    public void work() { //.... working much more }
}
class Superworker : Iworker {
    public void work() { //.... working much more }
    public void Eat() { //.... eating in lunch break }
}
class Manager {
    Iworkable worker;
    public void Setworker(Iworkable w) { worker=w; }
    public void Manage() { worker.work(); }
}
```

QA

Here is the reworked code supporting the Interface Segregation Principle. By splitting the IWorker interface in two different interfaces the new Robot class is no longer forced to implement the Eat() method. Also if we need another functionality for the robot like recharging we create another interface IRechargeble with a method Recharge().

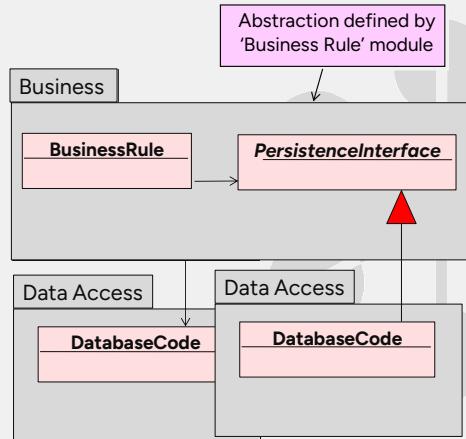
Like every principle, ISP is a principle which requires additional time and effort spent to apply it during the design time and it increases the complexity of code.

But the benefit is the creation of a flexible design. If you overdo it you will end up with a lot of interfaces with single methods So experience and common sense need to prevail when identifying the areas where extension of code is more likely to happen in the future.

Dependency Inversion Principle (DIP)

"High-Level modules should not depend on low-level modules – both should depend on abstractions"

"Abstractions should not depend on details. Details should depend on abstractions"



QA

In conventional application architecture, lower-level components are designed to be consumed by higher-level components which enable increasingly complex systems to be built. In this composition, higher-level components depend directly upon lower-level components to achieve some task. This dependency upon lower-level components limits the reuse opportunities of the higher-level components.

The goal of the dependency inversion principle is to decouple application glue code from application logic. Reusing low-level components (application logic) becomes easier and maintainability is increased. This is facilitated by the separation of high-level components and low-level components into separate packages/libraries, where interfaces defining the behaviour/services required by the high-level component are owned by, and exist within, the high-level component's 'package'. The implementation of the high-level component's interface by the low-level component requires that the low-level component package depend upon the high-level component for compilation, thus inverting the conventional dependency relationship. Various patterns are then employed to facilitate the run-time provisioning of the chosen low-level component implementation to the high-level component.

Dependency Inversion Principle (DIP) – violation example

```
// Dependency Inversion Principle - Bad example
class Worker {
    public void work() { // ....working }
}
class Manager {
    Worker worker;
    public void SetWorker(Worker w) {
        worker = w;
    }
    public void Manage() {
        worker.Work();
    }
}
// now add ...
class SuperWorker {
    public void work() { //.... working much more }
}
```

QA

This is an example which violates the Dependency Inversion Principle. We have a manager class which is a high-level class, and a low-level class called Worker. We might need to add new functionality to our application to model the changes in the company structure determined by the employment of new specialised workers. We might consider creating a new class SuperWorker for this.

The Manager class may be quite complex and now we have to change it in order to introduce the new SuperWorker. What are the disadvantages?

- We have to change the Manager class (it may be complex and will involve time and effort to make the changes)
- Some of the current functionality from the manager class might be affected.
- Significant unit testing will need to be redone

All these problems could take a lot of time to solve and they might introduce new errors in the old functionality. The situation would be different if the application had been designed following the Dependency Inversion Principle (see overleaf).

Dependency Inversion Principle (DIP) – solution

```
// Dependency Inversion Principle - Good example
interface Iworkable { public void work(); }
class Worker : Iworkable {
    public void work() { // ....working }
}
class Manager {
    Iworkable worker;
    public void Setworker(Iworkable w) {
        worker = w;
    }
    public void Manage() {
        worker.work();
    }
}
// now add ...
class Superworker : Iworkable {
    public void work() { //.... working much more }
}
```

QA

We design the Manager class, an IWorkable interface and the Worker class implementing the IWorkable interface. If/when we add the SuperWorker class all we have to do is implement the IWorkable interface for it. No additional changes in the existing classes.

The code above supports the Dependency Inversion Principle.

In this new design the IWorkable Interface is the abstraction layer.

The abstraction defining the behaviour/services required by the high-level component are owned by, and would exist within the high-level component's 'package'. The implementation of the high-level component's interface by the low level component requires that the low-level component package depend upon the high-level component for compilation, thus inverting the conventional dependency relationship.

When this principle is applied it means the high-level classes are not working directly with low-level classes, they are using interfaces as an abstract layer. In this case, instantiation of new low-level objects inside the high-level classes(if necessary) can not be done using the 'new' operator. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory or Prototype.

Now, the problems seen previously are solved (assuming there is no change in the high level logic):

- Manager class doesn't require changes when adding SuperWorkers.
- Minimised risk to affect old functionality present in Manager class since we don't change it.
- No need to redo the unit testing for Manager class

Of course, using this principle means an increase in effort, which will result in more classes and interfaces to maintain and slightly more complex code. However the benefit will be greater flexibility. The technique should not be applied blindly nor should it be used in every case. If class functionality is likely to remain unchanged in the future, there is no need to apply this principle.

Design Patterns embody principles

They are a great way of grasping and implementing principles in your design/code

You have to strike a balance between:



Too simple
Easier to hack than follow the design
Rigid/Fragile/Repetitive etc.



Needless Complexity



→ How do you find that balance? – next slide...

QA

...By Unit Testing

Expose software that ought to be de-coupled

We naturally put responsibilities together

- Software design is largely about separating the responsibilities that are subject to change
- You don't want to do it if the change never happens
- When you do have to do it, you want to do this just once
 - I.e. take the first bullet then refactor so the following bullets don't hurt
 - Never say "we'll come back and fix that later"
 - Agile design is a process, not an event

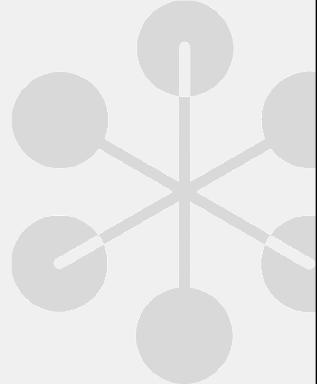
Unit Test is probably the best tool we have for driving good design



QA

Common Design Patterns

- Façade
- Composite
- Command
- Observer (subsumed into .NET as events)
- Strategy
 - Defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable within that family
- Factory
 - Creates objects without exposing instantiation logic to client
- Singleton
 - A class self-instantiates only instance



More Later...

QA

Further study and experience needed.

Hands on Lab

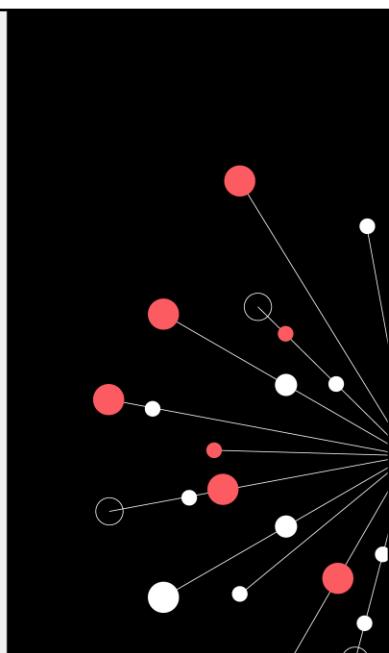
05 Solid Principles - Book Store Inventory System

Objective:

Your goal is to refactor code so that adheres to SOLID principles.

You are provided with a "Starter" program that manages an inventory system for a bookstore. The system allows adding books and CDs to the inventory and calculating the total stock value. The original code violates multiple SOLID principles. Your task is to refactor this code to make it more modular, maintainable, and extensible.

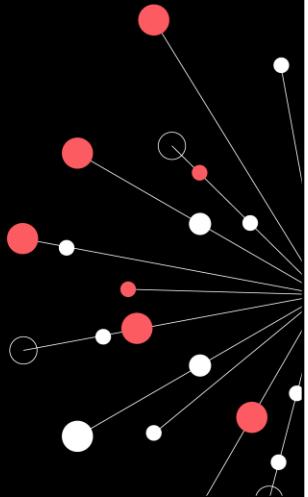
QA



Review

- Symptoms of a degrading design
 - Rigidity, fragility, immobility, viscosity
- Degradation of dependency architecture
- SOLID
 - SRP / OCP / LSP / ISP / DIP

QA



05 Design Patterns

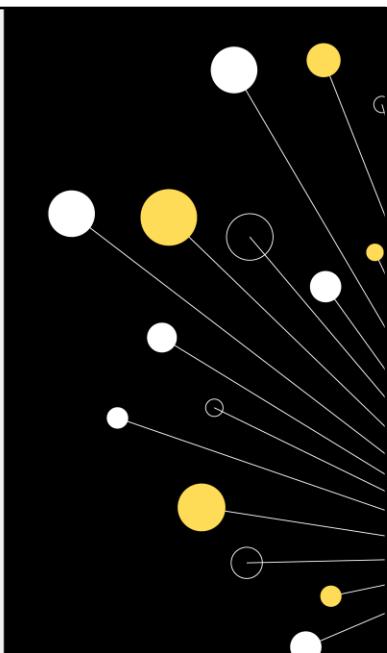
C# for HAAS F1



Learning objectives

Provide an introduction to coding design patterns. To understand their benefits and how to implement them and to gain exposure to a number of classic patterns.

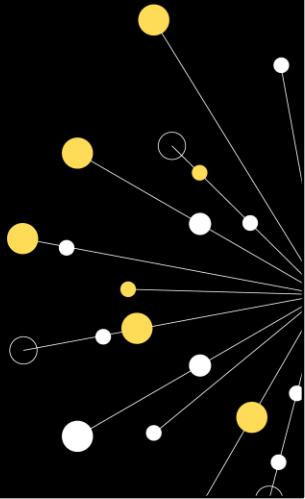
QA



Session Content

- Introduction to Design Patterns
- Pattern Classifications
- Creational Patterns
 - Singleton, Factory, Builder, Prototype
- Structural Patterns
 - Adapter, Decorator, Composite, Facade
- Behavioural Patterns
 - Strategy, Observer, Command

QA

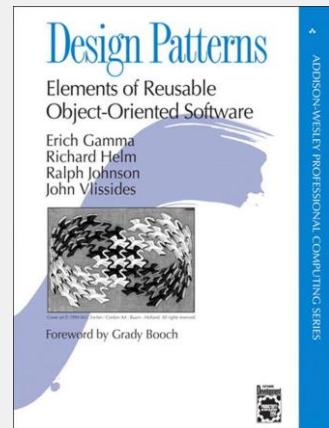


Design Patterns - What Are They?

WHAT?

Common solutions to problems which are regularly encountered in programming.

Created and collated by the "Gang of Four" in "Design Patterns" book.



QA

Design Patterns

Software design patterns are common solutions to problems which are regularly encountered in programming. These particular patterns deal with object-oriented programming exclusively. So, applying these patterns to any other programming paradigm is not a good idea. Some pattern proponents say that in the object-oriented world, these design patterns are full-fledged best practices. Others may not go quite as far as this, but all accept OO coding patterns are things that should be taken seriously. The patterns we'll be looking at originate from a book called, Design Patterns, Elements of Reusable Object-oriented Software, written by a group of authors who have come to be known as The Gang of Four or GOF.

Design Patterns - WHY?

Some of the benefits of using design patterns are:

- Saves time.
- There are many C# and .NET design patterns that we can use in our C# .NET projects.
- Promotes reusability
- Help to reduce the total cost of ownership (TCO) of the software product
- Lead to faster development.



QA

Design patterns are already defined and provides industry-standard approach to solve a recurring problem. So, it saves time if we sensibly use the design pattern.

There are many C# and .NET design patterns that we can use in our C# .NET projects.

Using design patterns promotes reusability, that leads to more robust and highly maintainable code.

Design patterns help to reduce the total cost of ownership (TCO) of the software product. Since design patterns are already defined, it makes our code easier to understand and debug.

Design patterns lead to faster development and new members of a team can understand them easily.

Pattern Classifications

Creational

- Singleton
- Factory
- Builder
- Prototype

Structural

- Adapter
- Decorator
- Composite
- Facade

Behavioural

- Strategy
- Observer
- Command



QA

According to the GOF, design patterns are classified in three groups: Creational, Structural, Behavioural. For example, singleton, factory, builder and prototype are creational design patterns; adapter, decorator, composite and facade are structural patterns, and strategy, observer and command patterns are behavioural patterns. In the following slides, we will start examining design patterns by looking how they can be implemented in C# code.

Creational Patterns

Deal with scenarios that involve the creation of an object or objects

- tries to create them in a manner that is appropriate to the situation.

Creating objects in a haphazard manner could result in:

- Design problems
- Over-complex design.

Creational patterns aim to solve this by:

- Separating a system from how its objects are created and composed.

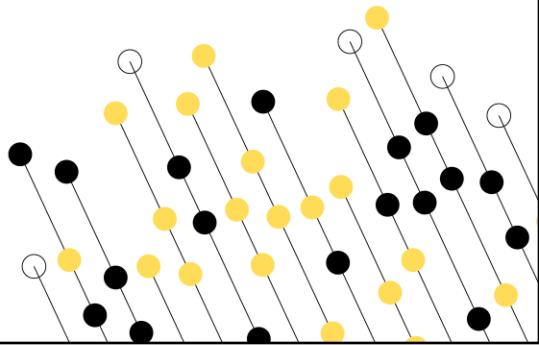


QA

Creational patterns deal with scenarios that involve the creation of an object or objects, trying to create them in a manner that is appropriate to the situation. By simply creating objects in a haphazard manner could result in design problems and/or an over-complex design. Creational patterns aim to solve this by separating a system from how its objects are created and composed.

Creational Pattern Classifications

- Singleton
- Factory
- Builder
- Prototype

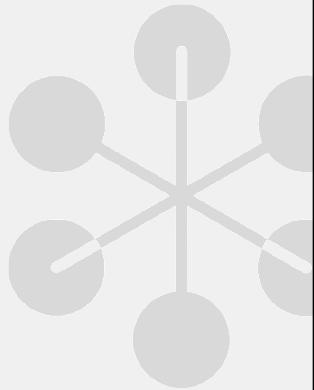


Added by Paul

The Singleton Pattern - Purpose

The purpose of the Singleton design pattern is to create only one instance of a class.

In other words, when an instance of a class is needed, if there is no instance, a new one is created. However, if an instance has already been created, the existing instance is used.

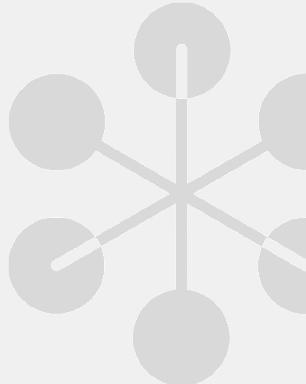


QA

Singleton Pattern - Implementation

The implementation of the Singleton pattern involves several standard steps:

- Make the class constructor private
- Create a private static variable that holds the single instance of the class.
- Provide a public static method that returns the instance of the class.



QA

Make the class constructor private to prevent other objects from using the new operator to create instances of the class.

Create a private static variable that holds the single instance of the class.

Provide a public static method that returns the instance of the class. This method often includes logic to create the instance if it hasn't been created yet, ensuring that the class is lazily instantiated.

Singleton Pattern Example Code



```
public sealed class Singleton {  
    private static Singleton instance = null;  
  
    private Singleton(){ }  
  
    public static Singleton Instance {  
        get {  
            if (instance == null)  
            {  
                instance = new Singleton();  
            }  
            return instance;  
        }  
    }  
}
```

QA

Singleton Pattern Thread Safe Example

In a multi-threaded environment, there is a possibility two threads could evaluate the test (if (instance == null) and find it to be true and then go on and create two instances.

Consequently, you may need to create a thread-safe version:

```
public sealed class Singleton {
    private static Singleton instance = null;
    private static readonly object padlock = new object();

    Singleton() { }

    public static Singleton Instance {
        get {
            lock (padlock) {
                if (instance == null) {
                    instance = new Singleton();
                }
                return instance;
            }
        }
    }
}
```

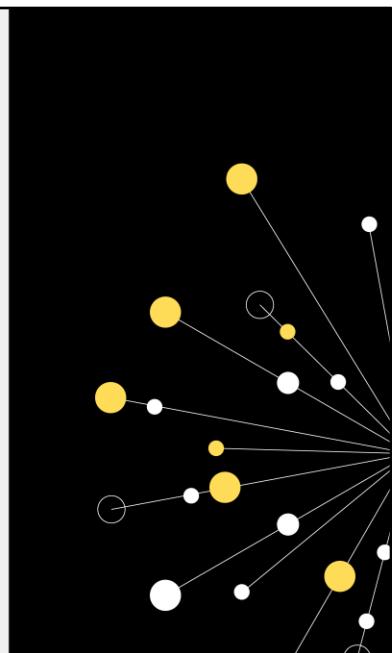
QA

You may find the following article taken from the csharpindepth.com that takes a (much) deeper dive into thread safety, lazy type initialisation and performance: [Implementing the Singleton Pattern in C# \(csharpindepth.com\)](http://csharpindepth.com/Articles/General/Implementing_the_Singleton_Pattern_in_C_.aspx). However, do note the underlying discussion was last updated in 2011 so things may have changed.

DEMO: Singleton Pattern

The example demonstrates how any consumer of the Singleton class will receive the same instance of Singleton, ensuring that global state managed by the singleton is shared across its consumers. This pattern is very effective for operations where actions need to be controlled from a single point, like caching, thread pools, or windows registry operations.

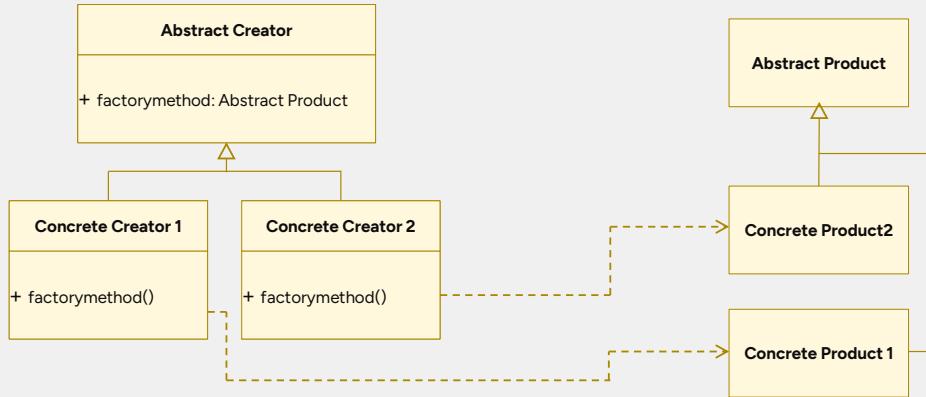
QA



Explanation of the Example

- The Singleton class has a private static variable instance which holds the only instance of the class.
- The constructor of the class is private, which prevents instantiation from outside of the class.
- The public static property Instance provides the global point of access to the singleton. The property ensures that the singleton instance is created only when it is first needed and subsequently returns the same instance for every subsequent request.
- The addition of the lock statement around the instantiation logic ensures that the class is thread-safe. This means if multiple threads try to get the instance at the same time, only one will be able to execute the instantiation code block, while the others will wait.

Factory Pattern - UML



QA

The abstract creator supports an interface that can generate and return an abstract product. The specific products produced (Concrete Product 1 and Concrete Product 2) are variations of this abstract product.

In the concrete factory creator class, the factory method is implemented as a pure virtual function with a return type of Abstract Product, enabling the return of any specific product derived from this abstract base.

The decision of what specific product the factory method returns is left to the concrete creators.

Factory Pattern Example Code

Factory Code

```
// Abstract Creator Class
public abstract class VehicleFactory {
    // Factory Method
    public abstract Vehicle CreateVehicle();
}

// Concrete Creator Class
public class CarFactory : VehicleFactory {
    public override Vehicle CreateVehicle() {
        return new Car();
    }
}

// Abstract 'Product' Class
public abstract class Vehicle{
    public abstract void Drive();
}

// 'Concrete 'Product' Class
public class Car : Vehicle {
    public override void Drive() {
        Console.WriteLine("driving a car");
    }
}
```

QA

Client Code

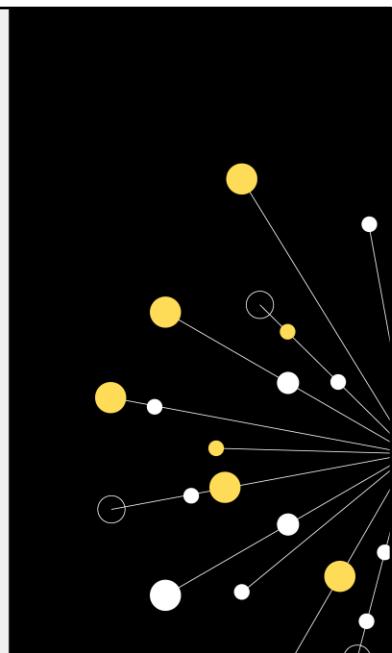
```
VehicleFactory carFactory = new CarFactory();
Vehicle myCar = carFactory.CreateVehicle();
myCar.Drive();

VehicleFactory lorryFactory = new LorryFactory();
Vehicle myLorry = lorryFactory.CreateVehicle();
myLorry.Drive();
```

DEMO: Factory Pattern

This structure allows the program to defer instantiation logic to derived classes, making the code more modular and easier to introduce new "products" without changing existing client code. The client (main program) only deals with the VehicleFactory and Vehicle abstract classes/interfaces. It does not know the details of the specific types of vehicles created. This is ideal for scenarios where you have multiple related products that may vary significantly but share common interfaces or base classes.

QA



Explanation of the Example

- Vehicle (Product) defines an interface for the type of object the factory method will create.
- Car and Lorry (ConcreteProducts) are different implementations of the Vehicle class.
- VehicleFactory (Creator) provides the factory method CreateVehicle that returns an object of type Vehicle. It is an abstract class.
- CarFactory and LorryFactory (ConcreteCreators) implement the CreateVehicle factory method to return instances of Car and Lorry, respectively.

The Abstract Factory Pattern

The Abstract Factory Pattern

- Offers an interface that allows families of related or dependent objects to be created without needing to specify their concrete classes.
- The pattern is used when you have interrelated objects that need to be created together.
- Uses a series of factory methods, one for each kind of product to be created



QA

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. The pattern is used when you have interrelated objects that need to be created together. The pattern is particularly useful when a system should be independent of how its products are created, composed, and represented. It involves a series of factory methods, one for each kind of product to be created.

The Factory vs. Abstract Factory Pattern

Are the Factory and Abstract Factory Pattern the Same?

In short, No though the Abstract Factory pattern does make use of the Factory pattern.

The Abstract Factory pattern allows each factory to create a suite of related products without having to understand their makeup. In contrast, the Factory Method only allows a factory to produce a single type of product



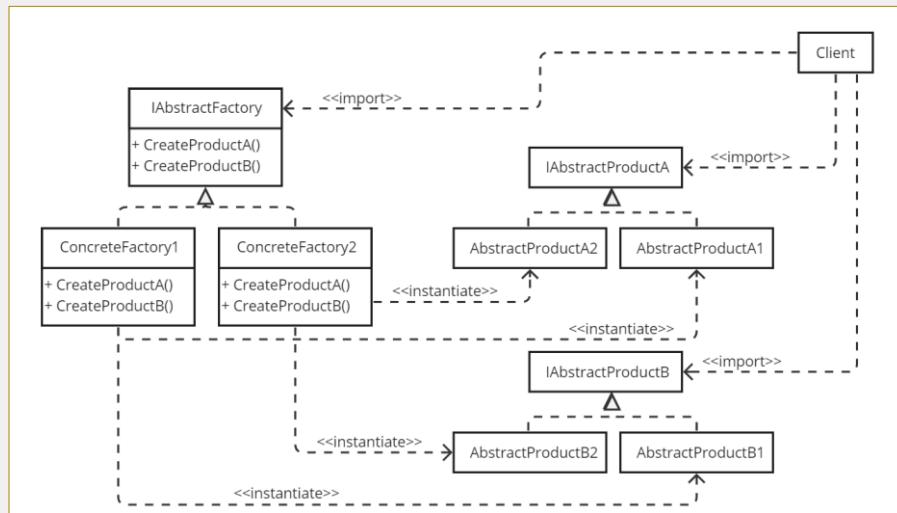
QA

Intent and Complexity: The Factory Method is about creating one product but deferring the instantiation logic to subclasses. Whereas the Abstract Factory pattern is used for creating families of related or dependent products without having to specify/understand their concrete classes.

Implementation: The Factory Method involves a single method to create products, whereas The Abstract Factory pattern uses multiple factory methods to create related products.

Use Case: Use Factory Method when you want to extend a single product type's instantiation logic. Use Abstract Factory when you need to create families of related products or need consistent usage among products that need to be used together.

Abstract Factory Pattern - UML



Abstract Factory Pattern Example Code

```
public interface IvehicleFactory{  
    ICar CreateCar();  
    ILorry CreateLorry();  
}  
  
public class CityVehicleFactory :  
    Ivehiclefactory {  
    public ICar CreateCar() {  
        return new CityCar();  
    }  
    public ILorry CreateLorry(){  
        return new CityLorry();  
    }  
}  
  
public class OutOfTownVehicleFactory :  
    Ivehiclefactory {  
    public ICar CreateCar() {  
        return new OffRoadCar();  
    }  
    public ILorry CreateLorry() {  
        return new IntercityLorry();  
    }  
}
```

```
public interface Icar {  
    void Drive();  
}  
  
public interface ILorry {  
    void LoadCargo();  
}  
  
public class CityCar : Icar {  
    public void Drive() {  
        Console.WriteLine("A city car is on the move");  
    }  
}  
  
public class CityLorry : ILorry {  
    public void Loadcargo() {  
        Console.WriteLine("City lorry cargo is loaded");  
    }  
}  
  
public class OffroadCar {...}  
Public class IntercityLorry {...}
```

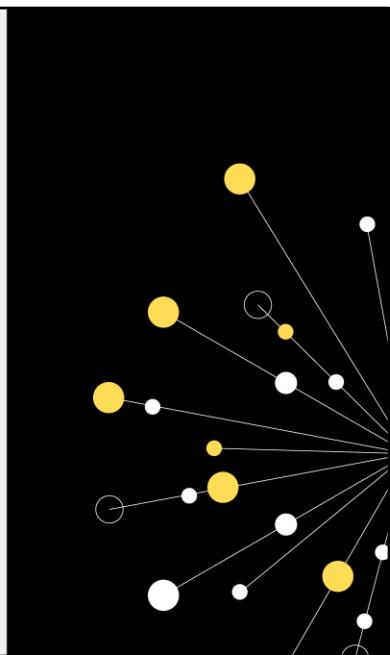
```
IvehicleFactory cityFactory = new CityVehicleFactory();           Client Code  
ICar citycar = cityFactory.CreateCar();  
ILorry cityLorry = cityFactory.CreateLorry();  
citycar.Drive();  
cityLorry.LoadCargo();  
  
// Create out of town vehicles  
IvehicleFactory outOfTownFactory = new OutOfTownVehicleFactory();  
ICar offroadCar = outOfTownFactory.CreateCar();  
ILorry offroadLorry = outOfTownFactory.CreateLorry();  
offroadCar.Drive();  
offroadLorry.LoadCargo();
```

QA

DEMO: The Abstract Factory Pattern

The design effectively demonstrates the Abstract Factory pattern, facilitating easy expansion and modification while maintaining the separation of concerns between the creation logic and the usage of products.

QA



Explanation of the Example

The IVehicleFactory interface acts as the abstract factory, with methods for creating both cars and trucks.

CityVehicleFactory and OffroadVehicleFactory are concrete factories that produce specific products (city or offroad vehicles).

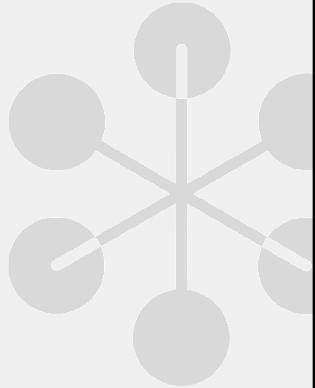
ICar and ITruck are abstract product interfaces, which have different concrete implementations based on the vehicle type (city or offroad).

The client uses the abstract factory to create abstract products. This way, the client is decoupled from the creation of the actual products, and adding new vehicle types or environments would not require changes to the client code.

The Builder Pattern - Implementation

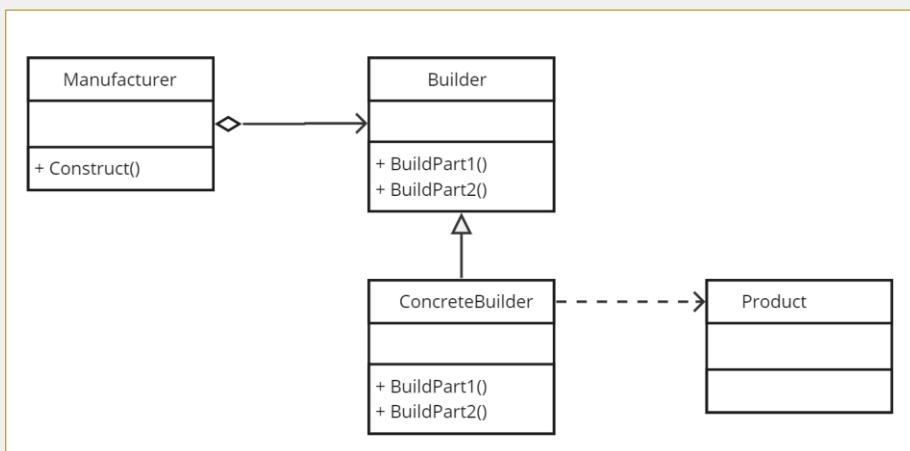
Components of the Builder Pattern

- **Builder:** Specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder:** Constructs and assembles parts of the product by implementing the Builder interface. It defines and keeps track of the representation it creates and provides an interface for retrieving the product.
- **Director:** Constructs an object using the Builder interface.
- **Product:** Represents the complex object being built



QA

Builder Pattern - UML



QA

The Builder Pattern Example Code - 1

Product Code

```
public class Car {  
    private List<string> _parts = new List<string>();  
    public Cartype Type { get; set; }  
  
    public void Add(string part) => _parts.Add(part);  
    public void Show(){  
        Console.WriteLine($"{this.Type} Car\n\tProduct Parts -----");  
        _parts.ForEach(part => Console.WriteLine($" \t{part}"));  
    }  
}
```

Builder Code

```
public abstract class CarBuilder {  
    protected Cartype cartype { get; init; }  
    protected Car car;  
    public Car Getcar() => car;  
    public void CreateNewCar() =>  
        car = new Car0 {Type = cartype};  
    public abstract void BuildChassis();  
    public abstract void BuildEngine();  
    public abstract void Buildwheels();  
    public abstract void Builddoors();  
}
```

Product Type Code

```
public enum Cartype{  
    Saloon,  
    Estate,  
    SUV,  
    Sports  
}
```

QA

The Builder Pattern Example Code - 2

Concrete Builder Code (Repeat for other car types)

```
public class EstateCarBuilder: CarBuilder {
    public EstateCarBuilder() => carType = CarType.Estate;

    public override void BuildChassis() => car.Add("Estate chassis");
    public override void BuildEngine() => car.Add("2000 cc");
    public override void BuildWheels() => car.Add("4 wheels");
    public override void BuildDoors() => car.Add("5 doors");
}
```

Manufacturer Code

```
public class Manufacturer {
    public void Construct(CarBuilder carBuilder) {
        carBuilder.CreateNewCar();
        carBuilder.BuildChassis();
        carBuilder.BuildEngine();
        carBuilder.BuildWheels();
        carBuilder.BuildDoors();
    }
}
```

QA

Client Code

```
static void Main(string[] args) {
    List<Car> cars = new List<Car>();
    Manufacturer manufacturer = new Manufacturer();

    CarBuilder builder = new SaloonCarBuilder();
    manufacturer.Construct(builder);
    Car car = builder.GetCar();
    cars.Add(car);

    builder = new EstateCarBuilder();
    manufacturer.Construct(builder);
    car = builder.GetCar();
    cars.Add(car);

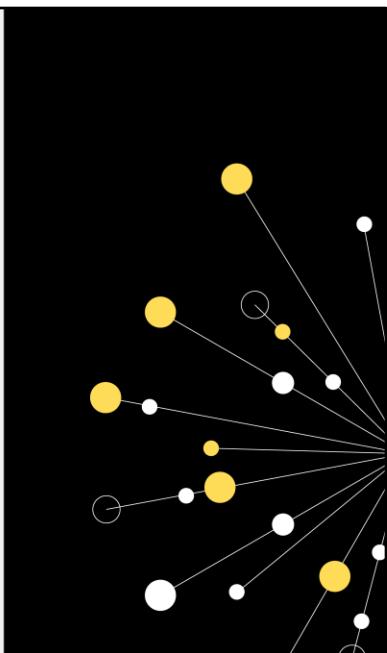
    cars.ForEach(c => c.Show());
}
```

DEMO: The Builder Pattern

The client (Program class) creates a Director object and a ConcreteBuilder object and passes the ConcreteBuilder to the Director. The director instructs the builder about what parts to build and in what order. After the building process, the builder returns the product to the client.

This approach allows different types of cars to be constructed using the same building process. It encapsulates the construction logic for a specific type of car within a ConcreteBuilder class, which can be replaced with a different ConcreteBuilder to construct a different type of car. This separation of concerns promotes reusability and flexibility.

QA



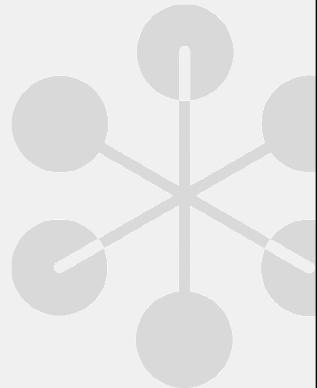
Explanation of the Example

- Car (Product) represents the complex object under construction.
- CarBuilder (Builder) provides an abstract interface for creating parts of the Car.
- EstateCarBuilder, SaloonCarBuilder, SportsCarBuilder and SUVCarBuilder (ConcreteBuilders) construct and assemble parts of the product implementing the CarBuilder interface. Each has a different representation of the car.
- Manufacturer (Director) constructs an object using the CarBuilder interface.

The Prototype Pattern - Purpose

The Prototype pattern is used when the type of created objects is determined by a prototypical instance, and the objects are created by this prototype.

- Useful when the creation of an object involves costly or complex operations.
- Serve a copy of an existing prototype instance rather than creating one from scratch
- This pattern helps in:
- Reducing the need for creating subclasses.
- Hiding the complexities of making new instances from the client.
- Optimizing performance when the initialization of a class is resource-intensive.



QA

The Prototype Pattern - Implementation

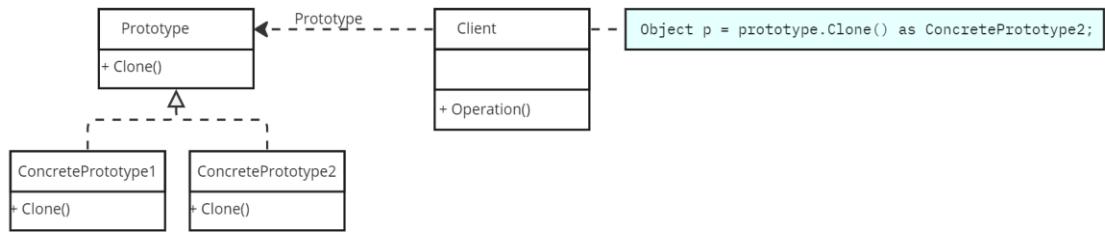
In C#, the Prototype pattern can be implemented using the **ICloneable** interface, which defines a method `Clone()` for cloning an object. It's important to know the difference between shallow and deep copying:

- **Shallow copy**: duplicates as little as possible. A shallow copy of an object is a new object whose instance variables are identical references to the corresponding instance variables in the original.
- **Deep copy**: duplicates everything. A deep copy of an object is a new object with entirely new instances of the duplicable elements of the object.

QA



Prototype Pattern - UML



QA

The Prototype Pattern Example Code

IPrototype Code

```
public interface IPrototype
{
    IPrototype Clone();
}
```

Client Code

```
ConcretePrototype prototype = new
ConcretePrototype("Sadia Saleem", 30);
ConcretePrototype clone = prototype.Clone() as
ConcretePrototype;

// Display original and cloned object
prototype.DisplayInfo();
clone.DisplayInfo();

// Making changes to verify that deep copy is indeed
clone.Name = "Isabelle Necessary";
clone.Age = 25;

Console.WriteLine("After changes to cloned object:");
prototype.DisplayInfo();
clone.DisplayInfo();
```

ConcretePrototype Code

```
public class ConcretePrototype : IPrototype {
    public int Age { get; set; }
    public string Name { get; set; }

    // Constructor for initializing data
    public ConcretePrototype(string name, int age) {
        this.Name = name;
        this.Age = age;
    }

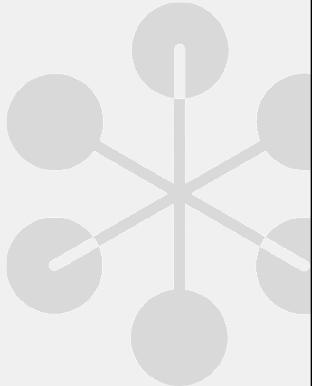
    // Copy Constructor for deep copying
    public ConcretePrototype(ConcretePrototype prototype) {
        Name = prototype.Name;
        Age = prototype.Age;
    }

    // Cloning method
    public IPrototype Clone() => new ConcretePrototype(this);
    public void DisplayInfo() =>
        Console.WriteLine($"Name: {Name}, Age: {Age}");
}
```

QA

DEMO: The Prototype Pattern

The example demonstrates the flexibility of the Prototype pattern, enabling easy and efficient creation of new objects by copying existing instances. This can lead to performance improvements in systems where object creation is a bottleneck.



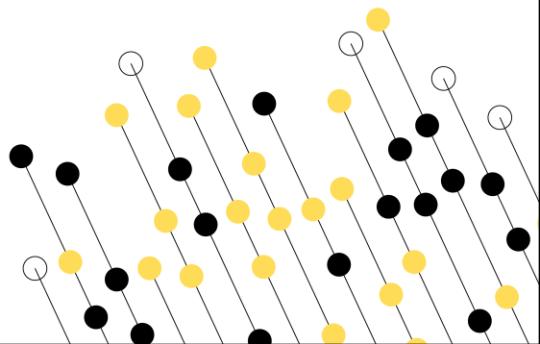
QA

Explanation of the Example

- The IPrototype interface with a Clone() method for cloning objects.
- ConcretePrototype implements this interface and includes a copy constructor for creating a deep copy of itself.
- The Main() method creates an instance of ConcretePrototype, uses the Clone() method to create a deep copy, and changes the properties of the cloned object to demonstrate that the original object remains unaffected, proving the deep copy.

Structural Pattern Classifications

- Adapter
- Decorator
- Composite
- Facade

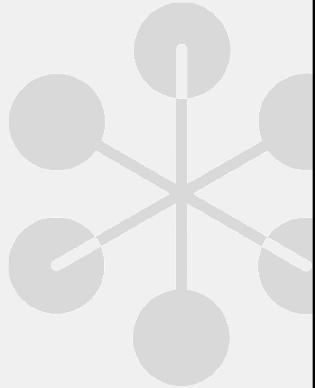


Slide added by Paul

The Adapter Pattern - Implementation

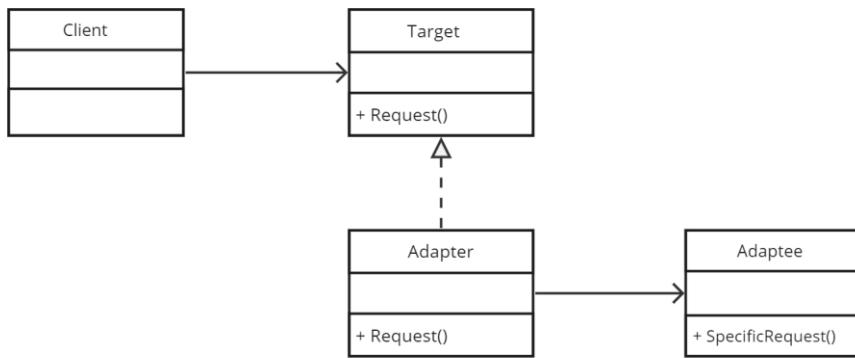
There are two primary ways to implement the Adapter pattern one of which can't be used in C#:

- **Object Adapter:** Uses composition to contain the adaptee in the adapter object. This approach uses the adapter's instance variables to hold a reference to an instance of the adaptee.
- **Class Adapter:** Uses multiple inheritance to adapt one interface to another (**not applicable in C#** due to its lack of support for multiple inheritance at the class level).



QA

Adapter Pattern - UML



QA

The Adapter Pattern Example Code

ITarget Code

```
public interface Itarget {  
    void Request();  
}
```

Adapter Code

```
// The 'Adapter' class implements the Itarget  
// interface and wraps an instance of  
// the Adaptee class.  
  
public class Adapter: Itarget {  
    private Adaptee _adaptee;  
  
    public Adapter(Adaptee adaptee) =>  
        _adaptee = adaptee;  
  
    // Possibly doing some other work  
    // and then adapting  
    // the Adaptee's SpecificRequest to  
    // the Target's Request.  
    public void Request() =>  
        _adaptee.SpecificRequest();  
}
```

Adaptee Code

```
// The 'Adaptee' class contains some useful behavior,  
// but its interface is incompatible with the existing  
// client code. The Adaptee needs some adaptation  
// before the client code can use it.  
public class Adaptee {  
    public void SpecificRequest() =>  
        Console.WriteLine("Called SpecificRequest()");  
}
```

Client Code

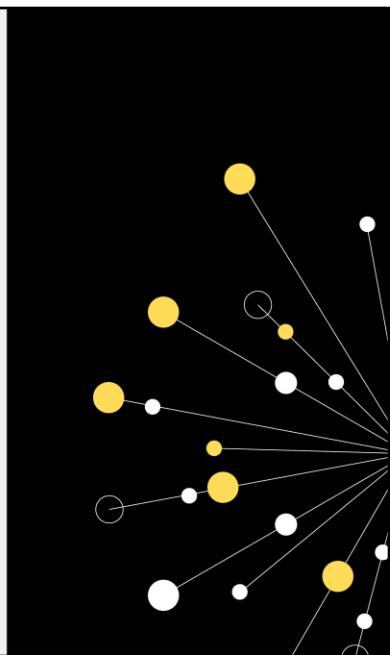
```
public void Main() {  
    // Adaptee's methods used via the Target interface.  
    Adaptee adaptee = new Adaptee();  
    ITarget target = new Adapter(adaptee);  
    target.Request();  
}
```

QA

DEMO: The Adapter Pattern

By using the Adapter pattern, existing classes with incompatible interfaces can be made to work together without modifying their source code, thus promoting reusability and flexibility within the system.

QA



Explanation of the Example

- ITarget is the interface that the client uses.
- Adaptee is the class that has some specific functionality (SpecificRequest), but its interface is not what the client expects or can use.
- Adapter is the class that implements the ITarget interface, holds a reference to an Adaptee object, and translates ITarget requests into Adaptee specific calls. This translation is usually one-to-one but can involve some transformation of the data passed between the Adapter and the Adaptee.
- Client uses the Target interface (ITarget) to call the Adaptee's method through the Adapter. The client is completely decoupled from the implementation of the adaptee, only knowing about the ITarget interface.

The Decorator Pattern - Purpose

Adds additional features or behaviours to an instance of a class while not modifying the other instances.

Flexible alternative to subclassing for extending functionality.

Solves problems such as

- How can a class be reused if it doesn't have an interface that a client needs?
- How can classes with incompatible interfaces work together?
- How can a class be given an alternative interface?



QA

The decorator design pattern is used to add additional features or behaviours to a particular instance of a class while not modifying any other instances of the same class. Decorators provide a flexible alternative to subclassing for extending functionality. It is important to fully understand the decorator pattern because once you know the techniques of decorating, you'll be able to give your or someone else's objects new responsibilities without making any code changes to the underlying classes.

The Decorator Pattern - Implementation

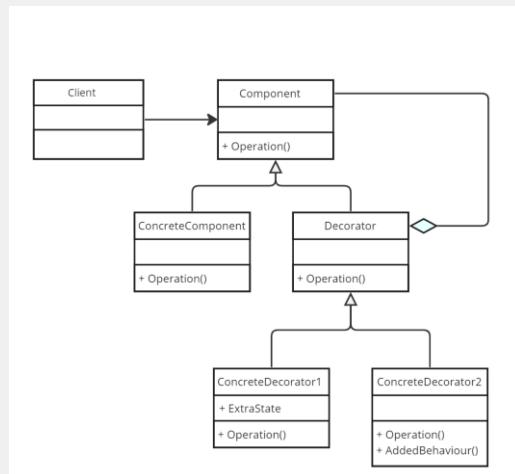
The Decorator pattern typically involves the following roles:

- **Component:** An interface designed for objects to which responsibilities can be dynamically assigned..
- **ConcreteComponent:** Describes an object that can have extra responsibilities attached to it.
- **Decorator:** Holds a reference to a Component object and provides an interface that matches the interface of the Component.
- **ConcreteDecorator:** Gives the component additional responsibilities.

QA



Decorator Pattern - UML



QA

The Decorator Pattern Example Code

Component Code

```
public abstract class Beverage {  
    public abstract string Description { get; }  
    public abstract decimal Cost { get; }  
}
```

ConcreteComponent Code

```
public class Espresso : Beverage {  
    public override string Description { get; } = "Espresso";  
    public override decimal Cost { get; } = 2.99m;  
}
```

Decorator Code

```
public abstract class CondimentDecorator : Beverage  
{  
    public abstract override string Description { get; }  
}
```

ConcreteDecorator Code

```
public class SteamedMilk : CondimentDecorator {  
    private Beverage _bev;  
    public SteamedMilk(Beverage bev) => _beverage = bev;  
  
    public override string Description {  
        get => _bev.Description + ", Steamed Milk";  
    }  
  
    public override decimal Cost {  
        get => _bev.Cost + 0.50m;  
    }  
}
```

Client Code

```
public void Main() {  
    List<Beverage> beverages = new List<Beverage>();  
  
    Beverage beverage;  
    beverage = new Espresso();  
    beverages.Add(beverage);  
  
    beverage = new Espresso();  
    beverage = new SteamedMilk(beverage);  
    beverages.Add(beverage);  
  
    beverages.ForEach(beverage =>  
        Console.WriteLine($"{beverage.Description}  
        {beverage.Cost:$0.00}"));  
}
```

QA

DEMO: The Decorator Pattern

This example of a coffee ordering system demonstrates how you can dynamically add functionality to an object without altering its structure. When the functionality of a component needs to be extended, the decorator pattern can be used as a flexible alternative to subclassing.



QA

Explanation of the Example

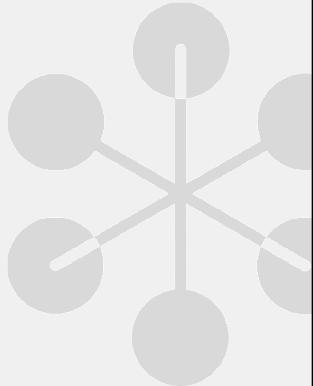
- Beverage (Component) defines the interface for objects that can have responsibilities added to them dynamically.
- Espresso (ConcreteComponent) defines an object (Espresso) to which additional responsibilities can be attached.
- CondimentDecorator (Decorator) maintains a reference to a Beverage object and defines an interface that conforms to Beverage's interface.
- ChocolateSauce and SteamedMilk (ConcreteDecorators) add responsibilities to the Beverage objects they decorate. They modify the behaviour of these objects by adding their own behaviour before or after delegating the call to the object they decorate.

The Composite Pattern - Purpose

The Composite pattern allows you to compose objects into tree-like structures to represent hierarchies where a group of objects is treated in the same way as a single instance of the same object type.

This pattern is particularly useful when:

- You need to implement a hierarchical structure.
- You want clients to ignore the differences between compositions of objects and individual objects.



QA

The Composite pattern allows you to compose objects into tree-like structures to represent hierarchies where a group of objects is treated in the same way as a single instance of the same object type. This pattern creates a uniform interface for individual objects and groups of objects enabling clients to treat individual objects and compositions of objects uniformly.

The Composite Pattern - Implementation

The composite pattern involves the following:

Component:

- This is an abstract class or interface with common operations for managing children .
- It usually includes methods like Add, Remove, and GetChild along with business logic functionality.

Leaf:

- Represents end objects in a composition.
- A leaf can't have any children.

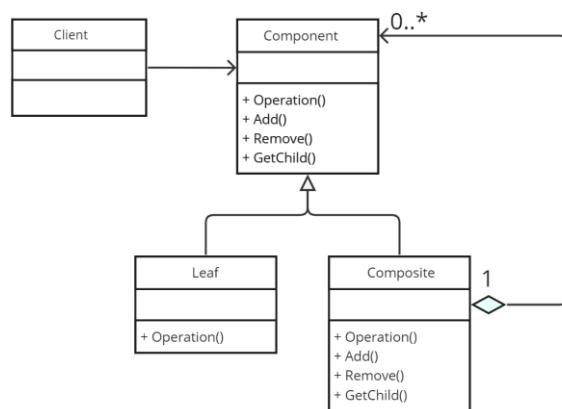
Composite:

- Defines behaviour for components having children.
- Stores child components and implements child-related operations in the Component interface.



QA

Composite Pattern - UML



QA

The Composite Pattern Example Code

Component Code

```
public abstract class Component {  
    protected string name;  
    public Component(string name) => this.name = name;  
    public abstract void Add(Component component);  
    public abstract void Remove(Component component);  
    public abstract void Display(int depth);  
}
```

Composite Code

```
public class Composite : Component {  
    private List<Component> _children =  
        new List<Component>();  
  
    public Composite(string name) : base(name) {}  
  
    public override void Add(Component component) =>  
        _children.Add(component);  
  
    public override void Remove(Component component) =>  
        _children.Remove(component);  
  
    public override void Display(int depth) {  
        Console.WriteLine(new String('-', depth) + name);  
        _children.ForEach(comp =>  
            comp.Display(depth + 2));  
    }  
}
```

Leaf Code

```
public class Leaf : Component {  
    public Leaf(string name) : base(name) {}  
    public override void Add(Component component) =>  
        Console.WriteLine("Cannot add to a leaf");  
  
    public override void Remove(Component component) =>  
        Console.WriteLine("Cannot remove from a leaf");  
  
    public override void Display(int depth) =>  
        Console.WriteLine(new String('-', depth) + name);  
}
```

Client Code

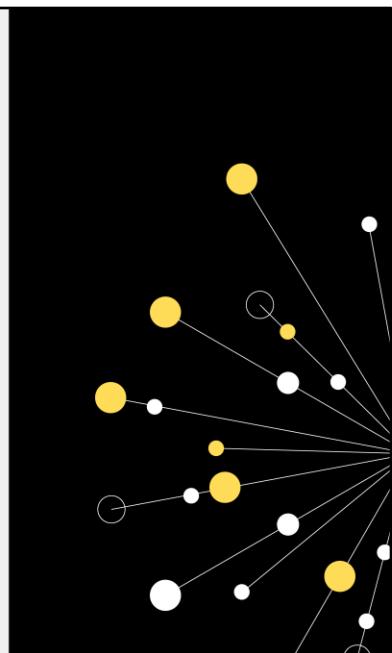
```
Composite root = new Composite("root");  
root.Add(new Leaf("Leaf A"));  
root.Add(new Leaf("Leaf B"));  
  
Composite comp = new Composite("Composite X");  
comp.Add(new Leaf("Leaf XA"));  
comp.Add(new Leaf("Leaf XB"));  
  
root.Add(comp);  
root.Add(new Leaf("Leaf C"));  
  
Leaf leaf = new Leaf("Leaf D");  
root.Add(leaf);  
root.Remove(leaf);  
  
root.Display(1);
```

QA

DEMO: The Composite Pattern

This structure allows you to work through the entire tree using a single interface, Component. Any operation like Display can be performed on either a single leaf or a complex composition of nodes, treating them uniformly. This encapsulation allows flexibility and ease of maintenance.

QA



Explanation of the Example

- The Component abstract class provides a common interface for all objects in the composition, both simple and complex.
- Leaf represents end objects in the structure. Leafs cannot have any children, so Add and Remove methods in Leaf class simply print a message indicating that these operations are not possible.
- Composite is the class that can have children (leaf nodes or other composites). It implements Add, Remove, and Display methods. The Display method prints an object and its children, recursively showing the structure.
- In the Main function, we create a tree structure with one root node (root), to which we add leaf nodes and a composite node (comp). We then manipulate the tree by adding and removing nodes, and finally, we call Display to show the structure.

The FAÇADE Pattern - Purpose

The Facade pattern provides a simplified interface to a more complicated subsystem.

- Does **NOT** encapsulate the subsystem.
- It provides a higher-level interface that makes the subsystem easier to use.
 - This is particularly useful when working with a large body of code, such as a complex library or API, where the direct management of objects within the subsystem can be complex or cumbersome.
- The facade simplifies the complexity behind the system, minimizing the communication and dependencies between systems.
 - Promotes loose coupling.
- Clients need only interact with the facade instead of the complex underlying system.



QA

The FAÇADE Pattern - Implementation

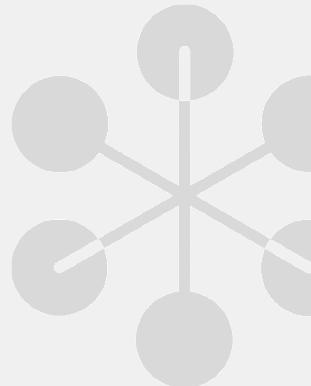
The Façade pattern involves the following:

Facade:

- Provides a simple interface to the complex logic of one or several subsystems.
- It delegates client requests to appropriate objects within the subsystem and manages their lifecycle.

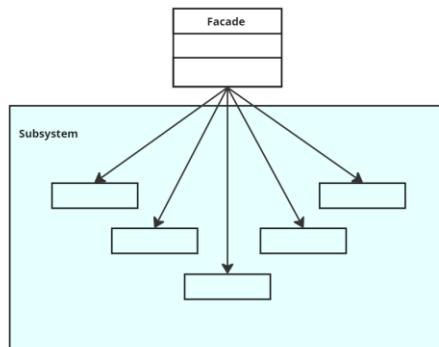
Subsystem Classes:

- Implement system functionality and perform the actual work.
- These classes are often fine-grained and intricate, requiring detailed knowledge to use correctly.



QA

Facade Pattern - UML



QA

The Facade Pattern Example Code – Part 1

DVD Code

```
// Subsystem Class 1
public class DvdPlayer {
    public void On() =>
        Console.WriteLine("DVD Player is on.");
    public void Play(string film) =>
        Console.WriteLine($"Playing \'{film}\'.");
    public void Off() =>
        Console.WriteLine("DVD Player is off.");
}
```

TheatreLights Code

```
// Subsystem Class 3
public class TheatreLights {
    public void Dim(int level) => console.WriteLine(
        $"Theatre lights dimmed to {level}%.");
    public void on() =>
        Console.WriteLine("Theatre lights are on.");
}
```

Projector Code

```
// Subsystem Class 2
public class Projector {
    public void On() =>
        Console.WriteLine("Projector is on.");
    public void SetInputDvd(DvdPlayer dvd) =>
        Console.WriteLine("Projector i/p set to DVD Player.");
    public void WideScreenMode() =>
        Console.WriteLine("Projector set to widescreen mode.");
    public void Off() =>
        Console.WriteLine("Projector is off.");
}
```

QA

The Facade Pattern Example Code – Part 2

Facade Code

```
public class HomeTheatreFacade {  
    private DvdPlayer _dvd;  
    private Projector _projector;  
    private TheatreLights _lights;  
  
    public HomeTheatreFacade(DvdPlayer dvd, Projector projector, TheatreLights lights) {  
        _dvd = dvd;  
        _projector = projector;  
        _lights = lights;  
    }  
  
    public void WatchFilm(string film) {  
        Console.WriteLine("Get ready to watch a Film...");  
        _lights.Dim(10);  
        _projector.On();  
        _projector.setInputDvd(_dvd);  
        _projector.WideScreenMode();  
        _dvd.On();  
        _dvd.Play(film);  
    }  
  
    public void EndFilm() {  
        Console.WriteLine("Shutting down the Film theatre...");  
        _dvd.Off();  
        _projector.Off();  
        _lights.On();  
    }  
}
```

QA

The Facade Pattern Example Code – Part 3

Client Code

```
static void Main(string[] args) {
    DvdPlayer dvd = new DvdPlayer();
    Projector projector = new Projector();
    TheatreLights lights = new TheatreLights();

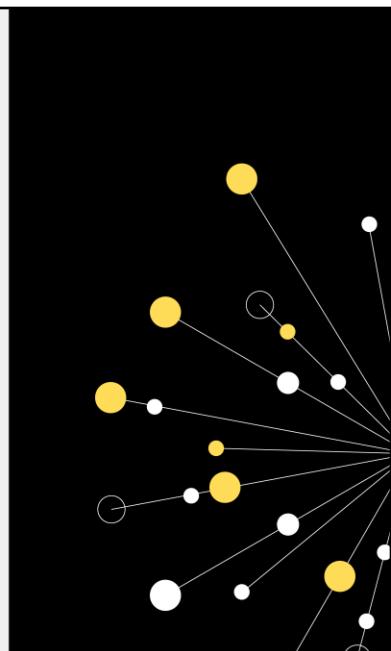
    HomeTheatreFacade homeTheater = new HomeTheatreFacade(dvd, projector, lights);
    homeTheater.WatchFilm("Postman Pat in 'The Heist!'");
    homeTheater.EndFilm();
}
```

QA

DEMO: The Facade Pattern

Clients interact with the facade instead of the subsystem directly, which makes the client code cleaner, easier to maintain, and less coupled to the inner workings of the subsystems. This setup embodies the essence of the Facade pattern by hiding the complexities of the subsystem and providing a simpler interface to the client.

QA



Explanation of the Example

- **Subsystem** classes (**DvdPlayer**, **Projector**, and **TheatreLights**) perform specific tasks and represent various parts of the home theatre system. Each class has a detailed interface that can be complex to manage directly.
- **HomeTheatreFacade** provides a simplified interface (**WatchMovie** and **EndMovie** methods) to the more complex parts of the home theatre system. It handles all the details of setting up for movie viewing and shutting down afterwards, which simplifies the client's responsibilities.

Behavioural Patterns

Focus on improving communication and responsibility distribution among objects

- Manage Complex Control Flows
- Encapsulate Requests
- Enhance Flexibility in Interaction
- Improve Scalability and Maintainability
- Promote Loose Coupling



QA

Behavioural Coding patterns are used to:

1. Manage Complex Control Flows: Behavioral patterns simplify complex control flows by managing and distributing responsibilities between various objects. This often involves defining a clear protocol or chain of responsibility which objects follow to handle a particular task.

2. Encapsulate Requests: These patterns can encapsulate information needed to perform actions or trigger events. This encapsulation allows for greater flexibility in specifying, queuing, and executing requests at different times.

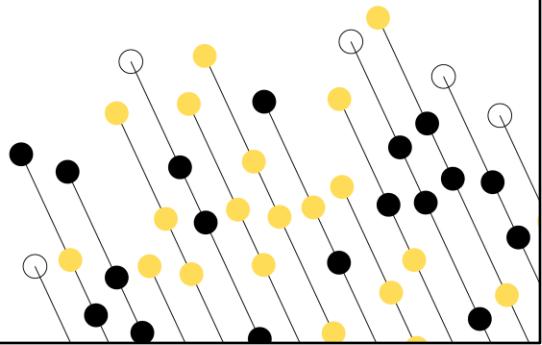
3. Enhance Flexibility in Interaction: By delegating how and when certain interactions occur between objects, behavioral patterns increase the flexibility and efficiency of communication.

4. Improve Scalability and Maintainability: Behavioral patterns can help make a system easier to scale and maintain by localizing changes to the behavior in a few objects or through standardized interfaces, without necessitating widespread modifications throughout the codebase.

5. Promote Loose Coupling: Many behavioral patterns aim to reduce dependencies between objects, which promotes loose coupling and fewer direct relationships. This makes systems easier to modify and extend.

Behavioural Pattern Classifications

- Strategy
- Observer
- Command

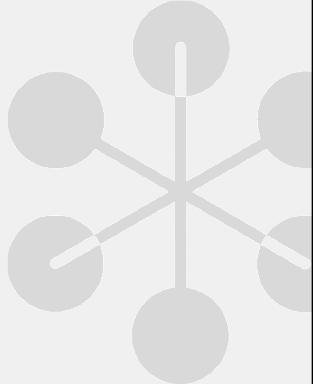


The Strategy Pattern - Purpose

The Strategy pattern allows algorithms to be dynamically selected at runtime.

The main idea behind the Strategy pattern is to:

- Define a family of algorithms
- Encapsulate each one as an object
- Make them interchangeable.
- Allow the chosen algorithm to vary independent of the clients that use it.
- The Strategy pattern is typically used to:
 - Manage algorithms, relationships, and responsibilities between objects.
 - Decouple the class operations from the behaviour that may vary often.
 - Provide alternative ways of executing the same task.



QA

The Strategy pattern allows algorithms to be dynamically selected at runtime. Rather than implement a single algorithm directly, the code receives instructions at run-time that specify which from a set of algorithms to use.

The Strategy Pattern - Implementation

The Strategy pattern uses the following:

Context:

- Maintains a reference to one of the concrete strategies.
- Communicates with this object only via the strategy interface.

Strategy:

- Defines an interface common to all supported algorithms.
- Context uses this interface to call the algorithm defined by a ConcreteStrategy.

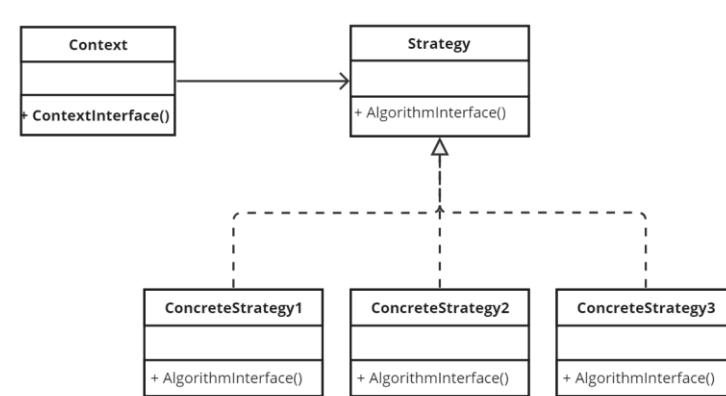
ConcreteStrategy:

- Implements the algorithm using the Strategy interface.



QA

Strategy Pattern - UML



QA

The Strategy Pattern Example Code – Part 1

Context Code

```
public class Context {  
    private IStrategy _strategy;  
    public Context(IStrategy strategy) =>  
        _strategy = strategy;  
    public void SetStrategy(IStrategy strategy) =>  
        _strategy = strategy;  
    public void ExecuteStrategy() =>  
        _strategy.Execute();  
}
```

IStrategy Code

```
public interface IPaymentStrategy {  
    void ProcessPayment(double amount);  
}
```

ConcreteStrategy1 Code

```
public class CreditCardPayment : IPaymentStrategy {  
    private string _cardNumber;  
    private string _cvv;  
    private string _expiryDate;  
  
    public CreditCardPayment(  
        string cardNumber, string cvv, string expiryDate){  
        _cardNumber = cardNumber;  
        _cvv = cvv;  
        _expiryDate = expiryDate;  
    }  
  
    public void ProcessPayment(double amount) {  
        Console.WriteLine(  
            $"Processing credit card payment for {amount:C}");  
        // Implementation for processing credit card payment  
    }  
}
```

QA

The Strategy Pattern Example Code – Part 2

ConcreteStrategy2 Code

```
public class PayPalPayment : IPaymentStrategy
{
    private string _emailAddress;
    private string _password;

    public PayPalPayment(string email, string password) {
        _emailAddress = email;
        _password = password;
    }

    public void ProcessPayment(double amount) {
        Console.WriteLine(
            $"Processing PayPal payment for {amount:c}");
        // Implementation for processing PayPal payment
    }
}
```

Client Code

```
static void Main(string[] args) {
    var paymentContext = new PaymentContext();

    var creditCardPayment =
        new CreditCardPayment("1234567890123456",
            "123", "12/25");
    paymentContext.SetPaymentStrategy(
        creditCardPayment);
    paymentContext.Pay(125.75);

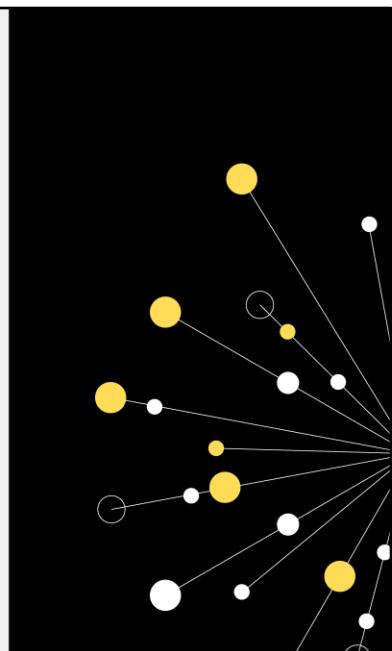
    var paypalPayment =
        new PayPalPayment("user@example.com",
            "password123");
    paymentContext.SetPaymentStrategy(
        paypalPayment);
    paymentContext.Pay(89.50)
}
```

QA

DEMO: The Strategy Pattern

This approach demonstrates the power of the Strategy pattern to facilitate adding new payment methods without changing the existing codebase significantly. It enhances the system's flexibility and makes it easier to extend. Each payment method can be developed, tested, and maintained independently, promoting better code organization and separation of concerns.

QA



Explanation of the Example

- **IPaymentStrategy:** Defines a common interface for all supported payment algorithms.
- **CreditCardPayment** and **PayPalPayment:** Implement the payment process for specific payment methods.
- **PaymentContext:** Uses a payment strategy to delegate the payment processing to the strategy object. It allows for changing the payment strategy dynamically depending on the user's choice.

The Observer Pattern - Purpose

The Observer pattern specifies one-to-many dependencies between objects:

- When an object's state changes, all its dependents are notified and updated automatically.

The pattern is useful in the following scenarios:

- When a change to one object requires changing others, and you don't necessarily know how many objects need to be changed.
- It helps to establish a subscription mechanism to send notifications to multiple objects about any events that happen to the object they're observing.



QA

The Observer pattern specifies one-to-many dependencies between objects:

- When an object's state changes, all its dependents are notified and updated automatically.
- The object that maintains the list of dependents (observers) is generally called the "subject" and its dependents are known as the "observers".

The pattern is useful in the following scenarios:

- When a change to one object requires changing others, and you don't necessarily know how many objects need to be changed.
- It helps to establish a subscription mechanism to send notifications to multiple objects about any events that happen to the object they're observing.

The Observer Pattern - Implementation

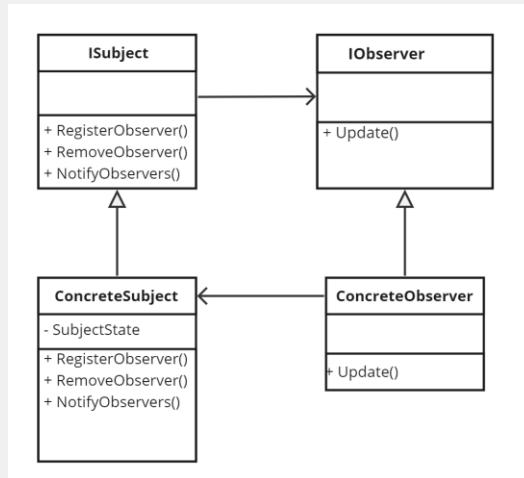
The Observer pattern uses the following:

- **Subject**
 - Manages a list of observers
 - Facilitates the addition and removal of observers.
- **Observer**
 - Provides an update interface for interested parties that need to be informed about changes to a subject.
- **Concrete Subject**
 - Stores relevant state for the ConcreteObserver.
 - When its state changes. It sends a notification to its observer(s).
- **Concrete Observer**
 - Maintains a reference to a ConcreteSubject object
 - Stores state that remains consistent with the state of the subject by implementing the Observer's updating interface.



QA

Observer Pattern - UML



QA

The Observer Pattern Example Code – Part 1

Subject Code

```
public interface Isubject {  
    void RegisterObserver(Iobserver observer);  
    void RemoveObserver(Iobserver observer);  
    void NotifyObservers();  
}
```

ConcreteSubject Code

```
public class WeatherDataSubject : Isubject {  
    private List<Iobserver> observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherDataSubject() =>  
        observers = new List<Iobserver>();  
    public void RegisterObserver(Iobserver observer) =>  
        observers.Add(observer);  
    public void RemoveObserver(Iobserver observer) =>  
        observers.Remove(observer);  
    public void NotifyObservers() {  
        observers.ForEach(o =>  
            o.Update(temperature, humidity, pressure));  
    }  
  
    public void MeasurementsChanged() => NotifyObservers();  
    public void SetMeasurements(float temp, float hum, float pres) {  
        this.temperature = temp;  
        this.humidity = hum;  
        this.pressure = pres;  
        MeasurementsChanged();  
    }  
}
```

QA

The Observer Pattern Example Code – Part 2

Observer Code

```
public interface Iobserver {  
    void Update(float temperature,  
               float humidity,  
               float pressure);  
}
```

Client Code

```
static void Main(string[] args) {  
    WeatherDataSubject wData =  
        new WeatherDataSubject();  
    CurrentConditionsObserver curDisplay = new  
        CurrentConditionsObserver(wData);  
  
    wData.SetMeasurements(27, 65, 1026f);  
    wData.SetMeasurements(28, 70, 1031f);  
}
```

ConcreteObserver Code

```
public class CurrentConditionsObserver : Iobserver {  
    private float temperature;  
    private float humidity;  
    private float pressure;  
    private ISubject weatherData;  
  
    public CurrentConditionsObserver(ISubject weatherData) {  
        this.weatherData = weatherData;  
        weatherData.RegisterObserver(this);  
    }  
  
    public void Update(  
        float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.pressure = pressure;  
        Display();  
    }  
  
    public void Display() {  
        Console.WriteLine(  
            $"Current conditions: {temperature}C degrees, " +  
            $"{humidity}% humidity and {pressure}hPa pressure");  
    }  
}
```

QA

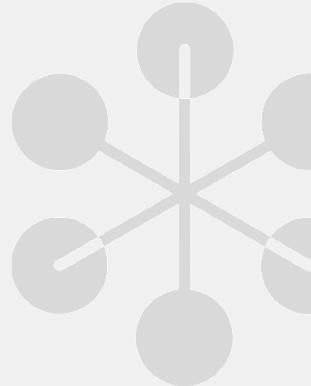
Observer Pattern Vs Delegates and Events

In .NET and C# you would normally use delegates and events to achieve the observer functionality

Delegates:

- Uses delegate and event to notify subscribers.
- Subscribers simply attach methods to the event, making it less verbose.
- Provides a more straightforward and idiomatic approach in C# for simpler use cases.

Both alternatives achieve the same functionality: when the State property changes, all registered observers/subscribers are notified.



QA

Delegate Example

Client Code

```
static void Main(string[] args) {
    var subject = new Subject();

    // Attach observers using delegates
    subject.StateChanged += (newState) =>
    {
        Console.WriteLine(
            $"Observer A update: State is now {newState}");
    };

    subject.StateChanged += (newState) =>
    {
        Console.WriteLine(
            $"Observer B update: State is now {newState}");
    };

    subject.State = 10; // Notifies all subscribers
    subject.State = 20; // Notifies all subscribers
}
```

ConcreteSubject Code

```
public class Subject
{
    public delegate void StatechangedHandler(int newState);
    public event StatechangedHandler StateChanged;

    private int state;
    public int State
    {
        get => state;
        set
        {
            state = value;
            OnStateChanged(state);
        }
    }

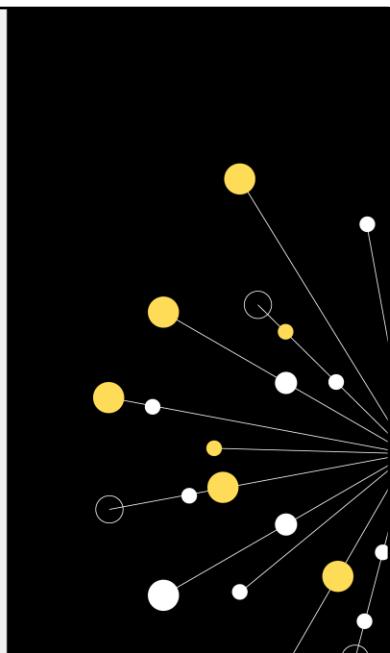
    protected virtual void OnStateChanged(int newState)
    {
        Statechanged?.Invoke(newState); // Notify all subscribers
    }
}
```

QA

DEMO: The Observer Pattern

The example effectively demonstrates how the Observer pattern can facilitate the real-time update of multiple displays based on a single data source, maintaining consistency across different components of the system. This pattern is also common in various other domains like GUI toolkits, event management systems, and more complex business applications where actions are triggered by changes in state or status.

QA



Explanation of the Example

- **WeatherData** is the Subject that maintains a list of observers (display devices) and notifies them of changes.
- **CurrentConditionsDisplay** is an Observer that updates itself whenever **WeatherData** changes.
- When **SetMeasurements** is called, it simulates new data coming from the weather station, which then calls **MeasurementsChanged**, triggering an update to all registered displays via **NotifyObservers**.

The Command Pattern - Purpose

The Command pattern turns a request into a standalone object that encapsulates **all** the information about the request that is required to perform a later action or trigger. Such as:

- Method name
- The object that owns the method
- Values for the method's parameters

This transformation lets you:

- Pass requests as method arguments
- Delay or queue request execution
- Manage undoable operations.
- Command pattern is a data-driven design pattern.



QA

Essentially, the Command pattern encapsulates a command request as an object, thereby letting you parameterize clients with queues, requests, and operations.

The Command Pattern - Implementation

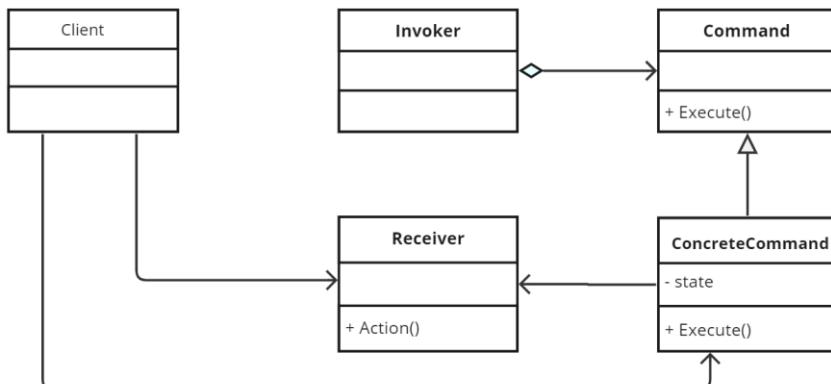
- **Command**
 - Specifies an interface for invoking an operation.
- **Concrete Command**
 - Extends the Command interface, implementing the invoking method by executing the Receiver's corresponding operation(s).
- **Receiver**
 - Knows how to carry out operations that are associated with a request. Any class can be a Receiver.
- **Invoker**
 - Triggers the command to deal with the request..
- **Client**
 - Specifies a ConcreteCommand object and sets its receiver.



QA

A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command. It's also known as action or transaction.

Command Pattern - UML



QA

The command object knows about the receiver and invokes methods of the receiver by supplying parameters. Values for parameters of the receiver's methods are stored in the command object. The receiver objects perform the job when the execute() method of command gets called. We pass this command object to an invoker object to perform the execute() method. Invoker object executes the method of the command object and passes the required parameters to it.

The Command Pattern Example Code – Part 1

ICommand Code

```
public interface ICommand {  
    void Execute();  
}
```

ConcreteCommand1 Code

```
public class LightOffCommand : ICommand {  
    private Light _light;  
    public LightOffCommand(Light light) => _light = light;  
    public void Execute() => _light.Turnoff();  
}
```

ConcreteCommand2 Code

```
public class LightOnCommand : ICommand {  
    private Light _light;  
    public LightOnCommand(Light light) => _light = light;  
    public void Execute() => _light.Turnon();  
}
```

Receiver Code

```
public class Light {  
    public LightStatus Status { get; set; } = LightStatus.Off;  
    public void TurnOn() {  
        if (Status == LightStatus.Off) {  
            Status = LightStatus.On;  
            Console.WriteLine("The light is on");  
        }  
    }  
  
    public void TurnOff() {  
        if (Status == LightStatus.On) {  
            Status = LightStatus.Off;  
            Console.WriteLine("The light is off");  
        }  
    }  
}
```

QA

The Command Pattern Example Code – Part 2

Invoker Code

```
public class RemoteControl {  
    private ICommand _onCommand;  
    private ICommand _offCommand;  
  
    public RemoteControl(ICommand onCommand, ICommand offCommand) {  
        _onCommand = onCommand;  
        _offCommand = offCommand;  
    }  
  
    public void PressOnButton() => _onCommand.Execute();  
    public void PressOffButton() => _offCommand.Execute();  
}
```

Client Code

```
static void Main(string[] args) {  
    Light light = new Light();  
    ICommand lighton = new LightOnCommand(light);  
    ICommand lightoff = new LightOffCommand(light);  
  
    RemoteControl control = new RemoteControl(lighton, lightoff);  
    control.PressOnButton(); // Output: The light is on  
    control.PressOffButton(); // Output: The light is off  
    control.PressOnButton(); // Output: NO OUTPUT  
    control.PressOnButton(); // Output: The light is on  
    control.PressOnButton(); // Output: NO OUTPUT  
}
```

QA

Command Pattern Vs Func Delegate

In .NET and C# you would normally use the Func delegate to achieve command pattern functionality

Func Delegate:

- Uses a lightweight approach where each operation is represented by a Func<string> delegate.
- Operations are directly defined as lambda expressions or methods and executed in sequence.
- Best suited for simpler scenarios where full encapsulation and decoupling are not required.

Both alternatives achieve the same functionality.

- The Command pattern provides a more structured and extensible approach,
- The Func delegate offers simplicity and conciseness.



QA

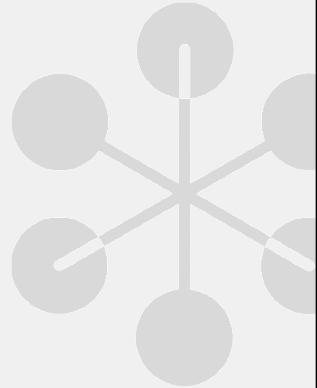
Func Delegate Example

```
Client Code
// Define a list of operations using Func
List<Func<string>> operations = new List<Func<string>>();

// Add operations
operations.Add(() =>
{
    int a = 5, b = 3;
    return $"Result of addition: {a + b}";
});

operations.Add(() =>
{
    int a = 5, b = 3;
    return $"Result of multiplication: {a * b}";
});

// Execute all operations
foreach (var operation in operations)
{
    Console.WriteLine(operation());
}
```

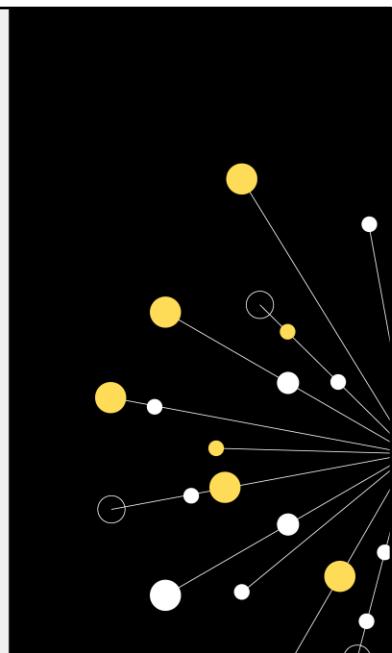


QA

DEMO: The Command Pattern

This example of switching a light on and off allows the invoker (a remote control) to be decoupled from the object handling the invocation (light switches), adding flexibility and extensibility to the framework. The Command pattern is very useful when you need to support undo operations, track a history of commands, or manage transactions and rollbacks.

QA



Explanation of the Example

- **Light** is the Receiver class. It knows how to perform the actual operations associated with turning a light on and off.
- **ICommand** is the Command interface with an Execute() method.
- **LightOnCommand** and **LightOffCommand** are ConcreteCommand classes that implement the ICommand interface and invoke operations on the Light.
- **RemoteControl** is the Invoker; it has a command object for each action (turning the light on and off) and it executes the command by calling the command's Execute() method when its button is pressed.
- **Program** class in Main() function acts as the Client, setting up the object relationships and starting commands.

Hands on Lab

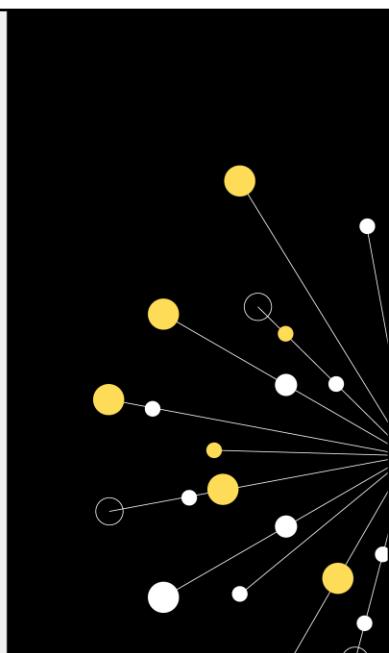
06 Design Patterns

Objective:

To create a basic vehicle management system that uses a number of design patterns.

You are tasked with designing and implementing a vehicle management system in C#. This system will allow users to create, manage, and monitor different types of vehicles such as cars, lorries, and motorcycles. To ensure the application is scalable, maintainable, and well-organized, you will employ the Factory, Composite, and Observer design patterns.

QA

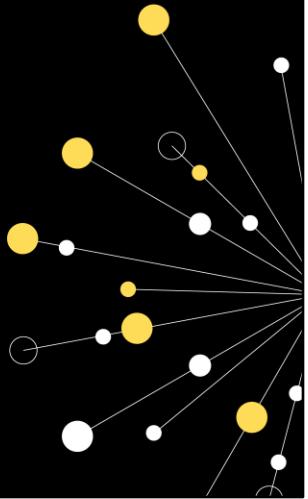


Summary

Know more about the configuration and use of Controllers and Actions

- Introduction to Design Patterns
- Pattern Classifications
- Creational Patterns
 - Singleton, Factory, Builder, Prototype
- Structural Patterns
- Adapter, Decorator, Composite, Facade
- Behavioural Patterns
- Strategy, Observer, Command

QA



07 JSON, CSV and Streams

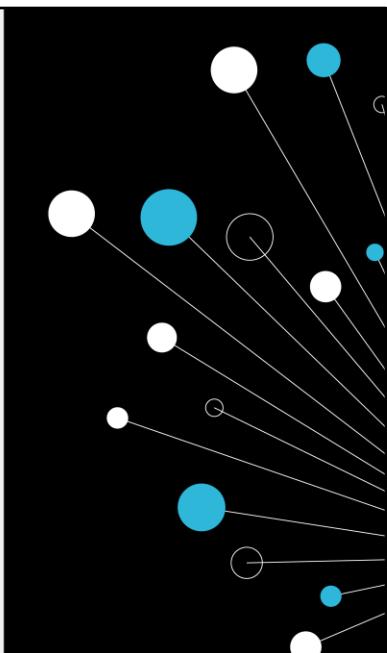
C# for HAAS F1



Learning objectives

- To understand how to use C# and .NET to read to and from different kinds of files (text, CSV, JSON) and streams

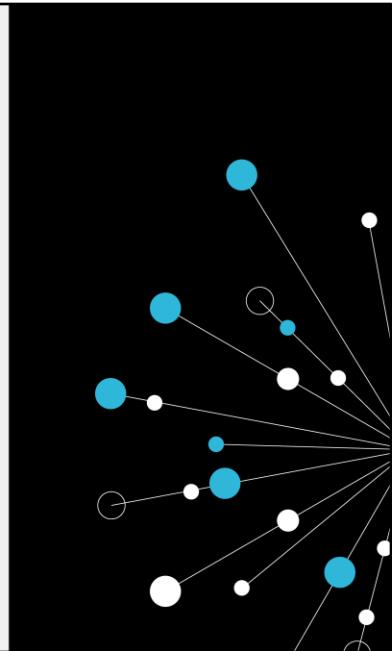
QA



Contents

- Accessing Files and Folders using the File System Classes
- Managing application data by using Reader and Writer classes
- Reading and Writing JSON
- Reading and Writing CSV files

QA



The System.IO Namespace

File and Directory classes

- Allow creation, deletion and manipulation of directories and files

StreamReader and StreamWriter

- Enable a program to access file contents as a stream of bytes or characters

FileStream

- Provide random access to files



QA

The System.IO namespace is important because it contains many classes that allow an application to perform input and output (I/O) operations in various ways through the file system. It also provides classes that allow an application to perform input and output operations on files and directories. The System.IO namespace is too big to be explained in detail here. However, some of the facilities available include:

The File and Directory classes that allow an application to create, delete, and manipulate directories and files.

The StreamReader and StreamWriter classes that enable a program to access file contents as a stream of bytes or characters.

The FileStream class that can be used to provide random access to files.

The BinaryReader and BinaryWriter classes that provide a way to read and write primitive data types as binary values.

File and Directory Classes

FileInfo, DirectoryInfo, DriveInfo and DriveType

- Expose system information about file system objects
- FileInfo & DirectoryInfo derive from FileSystemInfo but DriveInfo does not

File, Directory, Path

- Provide static methods to allow files, directories and paths to be manipulated

FileSystemInfo

- Enumerate files, directories and drives



QA

The System.IO namespace hosts a set of classes used to navigate and manipulate file, directories and drives. The file system classes are separated into two types of class: informational and utility

Most informational classes derive from FileSystemInfo. These classes expose all the system information about file system objects. However, the DriveInfo class does not derive from FileSystemInfo because although it is an informational class (like FileInfo and DirectoryInfo) it does not share the common sorts of behaviour (for example, you can delete files and directories but you cannot delete a drive).

The utility classes provide static methods to perform operations on file system objects.

Getting File Information

```
FileInfo mf = new FileInfo(@"C:\boot.ini");

if ( mf.Exists ) {
    Console.WriteLine("Filename: {0}", mf.Name);
    Console.WriteLine("Path: {0}", mf.FullName);
    Console.WriteLine("Length: {0}", mf.Length.ToString());
}
```

```
FileInfo fileToCopy = new FileInfo(@"C:\boot.ini");
fileToCopy.CopyTo(@"C:\copy of boot.ini");
```

QA

Unsurprisingly, the `FileInfo` class can be used to get information about a specified file! It is derived from `FileSystemInfo` and supports a number of methods and properties to that aim, some of which are listed below:

Properties

`CreationTime` Gets or sets the creation time of the current `FileSystemInfo` object.(inherited from `FileSystemInfo`)

`DirectoryName` Gets a string representing the directory's full path.

`Exists` Gets a value indicating whether a file exists.

`Extension` Gets the string representing the extension part of the file.(inherited from `FileSystemInfo`)

`FullName` Gets the full path of the directory or file.(inherited from `FileSystemInfo`)

`IsReadOnly` Gets or sets a value that determines if the current file is read only.

`LastAccessTime` Gets or sets the time the current file or directory was last accessed.(inherited from `FileSystemInfo`)

`Length` Gets the size of the current file.

`Name` Gets the name of the file.

Methods

`CopyTo` Copies an existing file to a new file.

`Create` Creates a file.

`Delete` Permanently deletes a file.

`MoveTo` Moves a specified file to a new location, providing the option to specify a new file name.

`Open` Opens a file with various read/write and sharing privileges.

`OpenRead` Creates a read-only `FileStream`.

`OpenText` Creates a `StreamReader` with UTF8 encoding that reads from an existing text file.

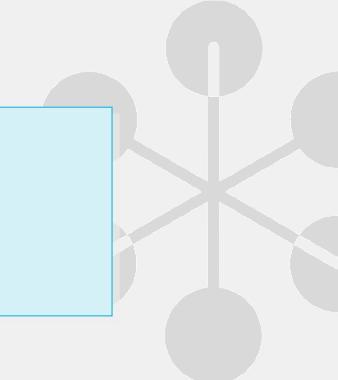
`OpenWrite` Creates a write-only `FileStream`.

`Replace` Replaces the contents of a specified file with the file described by the current `FileInfo` object, deleting the original file, and creating a backup of the replaced file.

DirectoryInfo

The DirectoryInfo class can be used to get information about directories and host an enumerable GetFiles collection

```
DirectoryInfo di = new DirectoryInfo(@"C:\windows");
Console.WriteLine("Directory: {0}", di.FullName);
foreach (FileInfo fi in di.GetFiles()) {
    Console.WriteLine("File: {0}", fi.Name);
}
```



QA

You equally won't be surprised to learn that the DirectoryInfo class can be used to get information about a specified directory! It is derived from FileSystemInfo and supports a number of methods and properties some of which are listed below:

Properties

Attributes Gets or sets the FileAttributes of the current FileSystemInfo.(inherited from FileSystemInfo)

CreationTime Gets or sets the creation time of the current FileSystemInfo object.(inherited from FileSystemInfo)

Exists Gets a value indicating whether the directory exists.

Extension Gets the string representing the extension part of the file.(inherited from FileSystemInfo)

FullName Gets the full path of the directory or file.(inherited from FileSystemInfo)

LastAccessTime Gets or sets the time the current file or directory was last accessed.(inherited from FileSystemInfo)

Name Gets the name of this DirectoryInfo instance.

Parent Gets the parent directory of a specified subdirectory.

Root Gets the root portion of a path.

Methods

Create Creates a directory.

CreateSubdirectory Creates a subdirectory or subdirectories on the specified path.
The specified path can be relative to this instance of the DirectoryInfo class.

Delete Deletes a DirectoryInfo and its contents from a path.

GetDirectories Returns the subdirectories of the current directory.

GetFiles Returns a file list from the current directory.

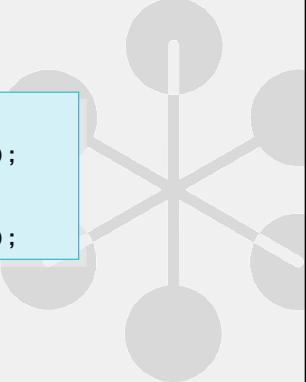
GetFileSystemInfos Retrieves an array of strongly typed FileSystemInfo objects representing files and subdirectories of the current directory.

MoveTo Moves a DirectoryInfo instance and its contents to a new path.

The Path Class

Allows the interrogation and parsing of individual parts of a file system path and the ability to change a file extension

```
string p = @"C:\boot.ini";
Console.WriteLine("Extension: {0}", Path.GetExtension(p));
Path.ChangeExtension(p, "bak");
Console.WriteLine("Extension: {0}", Path.GetExtension(p));
```



QA

The Path class provides methods for manipulating a file system path, some of which are listed below:

Methods

ChangeExtension Changes the extension of a path string. Note only the path string changes, not the actual file name extension.

Combine Combines two compatible path strings.

GetDirectoryName Returns the directory information for the specified path string.

GetExtension Returns the extension of the specified path string.

GetFileName Returns the file name and extension of the specified path string.

GetFullPath Returns the absolute path for the specified path string.

GetPathRoot Gets the root directory information of the specified path.

GetTempFileName Creates a uniquely named, zero-byte temporary file on disk and returns the full path of that file.

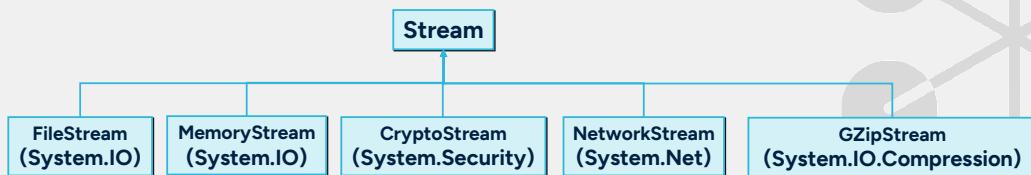
GetTempPath Returns the path of the current system's temporary folder.

HasExtension Determines whether a path includes a file name extension.

IsPathRooted Gets a value indicating whether the specified path string contains absolute or relative path information.

Streams

- Common way of dealing with sequential and random access to data within the .NET Framework
- Stream is an abstract base class



QA

A stream is an important concept in modern programming. Earlier programming languages such as Fortran, Cobol and Basic had input and output built in to those languages. Modern languages like C# tend to avoid having this and instead implement input and output via library functions. Output involves sending a stream of bytes to a device and whether that device is the screen, an object in memory, a file on disk, another computer on a network or a printer is unimportant. The same routines can be used with different destinations. In earlier languages you needed separate routines to handle each device. Streams make this flexibility possible.

Streams involve three fundamental operations:

- You can read from streams. Reading is the transfer of data from a stream into a data structure, such as an array of bytes.
- You can write to streams. Writing is the transfer of data from a data structure into a stream.
- Streams can support seeking. Seeking is the querying and modification of the current position within a stream. Seek capability depends on the kind of backing store a stream has. For example, network streams have no unified concept of a current position, and therefore typically do not support seeking.

Stream is the abstract base class of all streams.

Some thoughts before we see real IO Code

Reading/Writing files – you are interacting with the outside world

- File(name) might not exist, error situation, an ‘exception’ is thrown
- Open ‘file handles’ – an unmanaged resource (not Garbage Collected)
 - You are responsible for ‘closing’ files to free up resources
- This ‘must happen’ regardless of success / failure

C# has a try {...} finally {...} construct

- When something ‘must run regardless’ – learn it later in ‘Exceptions’
- finally blocks are ‘guaranteed’ to run (even with unhandled exception)
- But in IO code these can end up being nested (and messy)

C# designers invented a ‘using’ statement for making IO easier

- Not the ‘using’ directives that sit at the top of a file!
- IO is done via ‘using’ statements that compile into try / finallys



QA

We are about to go outside our cosy C# ‘Visual Studio’ world and interact with the file system.

Perhaps create ‘open file’ handles, these represent unmanaged windows resources that are not garbage collected. However, the Close() methods of the FCL IO classes will free up these resources.

So basically ‘Close’ has to happen, but it needs to happen if any sort of exception happens whilst the file is being processed. This could be an IO exception or any sort of exception thrown by your code whilst processing the data of a file.

The C# language as you will learn in a later ‘Exceptions’ chapter has a mechanism for ‘always’ running some code. It is called a finally block and it sits after a ‘try’ block.

```
try {  
..  
..    // exception might happen here  
} finally {  
..    // but this still runs enabling ‘cleanup to happen  
}
```

Unfortunately, when doing IO you are often working with multiple files and they all need closing so try / finally blocks can become nested and messy.

To solve this problem the language designers invented a ‘using’ statement that compiles into a try finally and can easily be nested.

Read on ..

The using statement – that simplifies code

```
using System.IO;
public void DoingIO() {
    using (StreamReader rdr = new StreamReader("in.txt")) {
        using (StreamWriter wtr = new StreamWriter("out.txt")) {
            string line;
            while ((line = rdr.ReadLine()) != null) {
                wtr.WriteLine(line);
            }
        }
    }
}
```

Using 'statement' in the code

Each 'using' compiles into
‘..do stuff..but always invoke Dispose().’

2 using's
needed here

Client code handles any IO Exception

```
using (StreamReader rdr1 = new StreamReader("in1.txt"),
       rdr2 = new StreamReader("in2.txt")) {
    // code that uses both rdr's...
}
```

When 2 refs of same type

QA

StreamReaders and StreamWriters provide basic functionality to read and write from and to a stream. The example above shows how you can perform a simple file copy operation.

To open a file for reading, the code in the example creates a new StreamReader object and passes the name of the file that needs to be opened in the constructor. Similarly, to open a file for writing, the example creates a new StreamWriter object and passes the output file name in its constructor. The example program then reads one line at a time from the input stream and writes that line to the output stream.

To free unmanaged resources (File handles etc) Close() or Dispose() can be called in a finally block. This can end up with some very messy try/try/try finally/finally/finally type code (covered in ‘Exceptions’ later) even. There is an implicit assumption here that any IO Exception wants to be passed back unhandled. Fortunately the language designers came up with an easy way of getting the compiler to generate the try / finally’s...the using statement.

You don’t need to worry about scope as you do with ret and finally ‘blocks’.

In the 1st example above ‘rdr’ is clearly in scope in the nested using. If you viewed this in ILDASM you would see

try...try ..finally (including Dispose() of wtr) ...finally (including Dispose() of rdr).

In the second example because both rdr1 and rdr2 are of same data type we can cope with one using block.

The File Class

Provides basic functionality to open file streams for reading and writing

```
using (FileStream fs = File.Open(@"C:\bootx.ini",
    FileMode.Open, FileAccess.Read)) {
    using (StreamReader sr = new StreamReader(fs)) {
        Console.WriteLine(sr.ReadToEnd());
    }
}
```

The OpenText method makes it simple to open a file for reading

```
using (StreamReader sr = File.OpenText(@"C:\boot.ini")) {
    Console.WriteLine(sr.ReadToEnd());
}
```

QA

The File class provides the basic functionality to open file streams for reading and writing. It supports a variety of Open methods:

Open, OpenText, OpenRead and OpenWrite

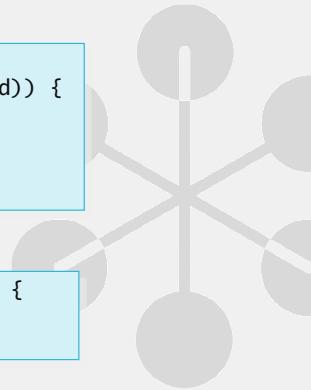
The FileMode enumeration is used to specify how a file is to be opened or created.

FileMode.Open opens an existing file. The FileAccess enumeration is used to determine the rights required when opening a table. FileAccess.Read is used to gain read-only access to its contents. If all you want to do is read out the entire file the ReadAllText method hides all the details of the stream and reader implementation:

Console.WriteLine(File.ReadAllText(@"C:\boot.ini"));

To write to a file you must open it for writing, the process being similar to opening a file for reading:

```
//Explicit FileStream
using(FileStream fs = File.Open(@"C:\myFile.txt", FileMode.Create,
FileAccess.Write)){
    using(StreamWriter sw = new StreamWriter(fs)) {
        sw.WriteLine("blah blah blah");
    }
}
//Implicit FileStream
using(StreamWriter sw = File.CreateText(@"C:\myOtherFile.txt")) {
    sw.WriteLine("more blah blah blah");
}
//
File.WriteAllText(@"C:\myFile.txt", "stuff");
```

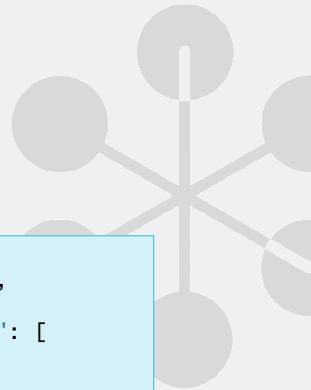


JSON

- stands for **JavaScript Object Notation**
- Lightweight format for storing and transporting data
- Frequently used when data is passed from a web server to a client
- Self-describing and easy to understand
- Use NuGet to install Newtonsoft.Json package

QA

```
{  
  "name": "Maria",  
  "age": 39,  
  "favouriteFilms": [  
    "Love Story",  
    "Inception",  
    "It's a wonderful Life"  
]  
}
```



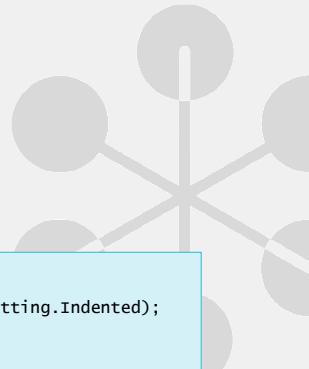
Serializing and Deserializing JSON

```
PersonFilms personFilms = new PersonFilms
{
    Name = "Andrew",
    Age = 50,
    FavouriteFilms = new List<string>() {
        "Toy Story",
        "Toy Story 2",
        "Aliens"
    }
};
```

```
using Newtonsoft.Json;

string pfs = JsonConvert.SerializeObject(personFilms, Newtonsoft.Json.Formatting.Indented);
Console.WriteLine(pfs);
PersonFilms pfs2 = JsonConvert.DeserializeObject<PersonFilms>(pfs);
Console.WriteLine(pfs2.Name);
Console.WriteLine(pfs2.Age);
pfs2.FavouriteFilms.ForEach(f => Console.WriteLine(f));
```

QA



JSON Anonymous Objects

```
var obj1 = new
{
    name = "Maria",
    age = 39,
    favouriteFilms = new List<string>() {
        "Love Story",
        "Inception",
        "It's a Wonderful Life"
    }
};

using Newtonsoft.Json;

string json1 = JsonConvert.SerializeObject(obj1, Newtonsoft.Json.Formatting.Indented);
File.WriteAllText("file1.json", json1);

// Reading JSON into an anonymous, dynamic object then picking out elements
string s1 = File.ReadAllText("file1.json");
dynamic obj2 = JsonConvert.DeserializeObject(s1);
Console.WriteLine(obj2.name);
Console.WriteLine(obj2.age);
Console.WriteLine(obj2.favouriteFilms);
```

QA

CSV

- A comma-separated value (CSV) file is a structured file where individual pieces of data are separated by commas.
- The first line of the file often contains "column" names
- Use NuGet to import CsvHelper package (other packages are available)

```
Title,ReleaseYear,Director,GrossRevenue  
Star Wars IV,1977,George Lucas,775  
Avatar,2009,James Cameron,2800  
Inception,2010,Christopher Nolan,837  
Interstellar,2014,Christopher Nolan,681  
Men in Black,1997,Barry Sonnenfeld,589
```



QA

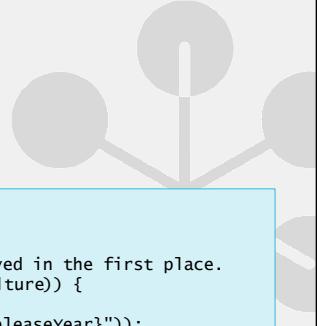
Reading a CSV File

```
public class Movie
{
    public string Title { get; set; }
    public int ReleaseYear { get; set; }
    public string Director { get; set; }
    public decimal? GrossRevenue { get; set; }
}

// Reading and displaying a list of people from a CSV file
using (var sr = new StreamReader("movies.csv"))

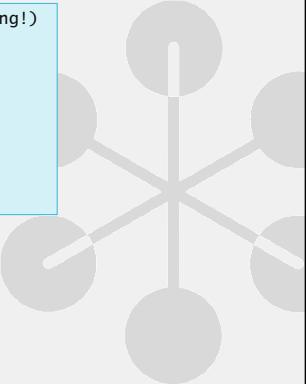
// InvariantCulture - I don't care, I don't want culture involved in the first place.
using (var reader = new CsvReader(sr, CultureInfo.InvariantCulture)) {
    var list = reader.GetRecords<Movie>().ToList();
    //list.ForEach(m => Console.WriteLine($"{m.Title} is {m.ReleaseYear}"));
    foreach (Movie m in list) {
        Console.WriteLine($"{m.Title} is {m.ReleaseYear}");
    }
}
```

QA



Reading a CSV File into a Dynamic Class

```
// Reading records into dynamic class (NB: every property value will be a string!)
using (var sr = new StreamReader("movies.csv"))
using (var reader = new CsvReader(sr, CultureInfo.InvariantCulture)) {
    var list = reader.GetRecords<dynamic>().ToList();
    foreach (var m in list) {
        Console.WriteLine($"{m.Title} is {m.ReleaseYear}");
    }
}
```



QA

Writing to a CSV File

```
// Writing a list to a csv file
var more_movies = new Movie[] {
    new Movie { Title = "2001: A Space Odyssey",
        ReleaseYear = 1968, Director="Stanley Kubrick", GrossRevenue=146 },
    new Movie { Title = "Dark Star",
        ReleaseYear = 1975, Director="John Carpenter", GrossRevenue=null },
    new Movie { Title = "The Martian",
        ReleaseYear = 2015, Director="Ridley Scott", GrossRevenue=631 }
};

using (var sw = new StreamWriter("updated_movies.csv"))
using (var writer = new CsvWriter(sw, CultureInfo.InvariantCulture)) {
    writer.WriteRecords(more_movies);
}
```



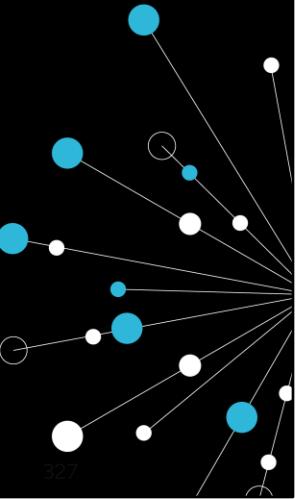
QA

Hands on Lab

Exercise 07 JSON, CSV and Streams

QA

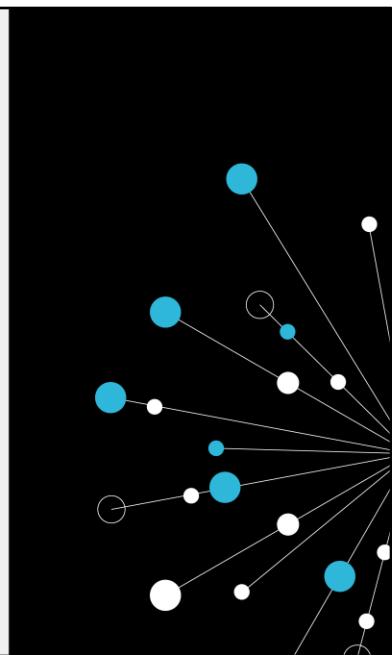
327



Summary

- DirectoryInfo, FileInfo, DriveInfo allow access to Directories, Files and Drives
- Use Streams to Read and Write data
- Reading and Writing JSON
- Reading and Writing CSV files

QA



08 Delegates and Lambdas

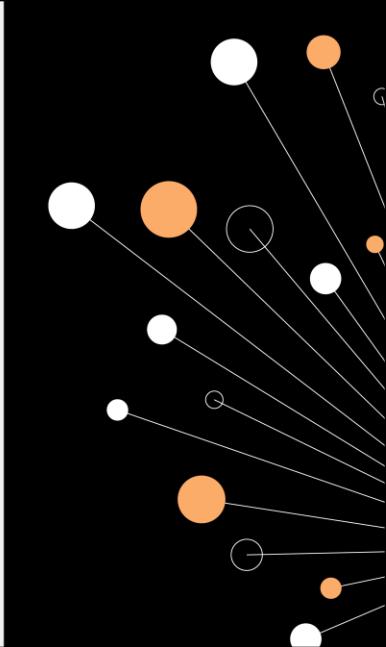
C# for HAAS F1



Learning objectives

- Delegates
- The Func delegate
- The Action delegate
- Lambda expressions
- The Predicate delegate
- Delegate and Lambda examples

QA



Delegates

- A **delegate** is a type that represents references to methods with a particular parameter list and return type
- You can instantiate a delegate and associate it with any method with a compatible signature and return type
- You invoke the method through the delegate instance
- Delegates are used to pass methods as arguments to other methods

Useful built-in delegate types are:

- Func
- Action



QA

The Func delegate

Func<> is a generic delegate which accepts zero or more input parameters and **one** return type.

The last parameter is the return type.

There are many overloads:

- **Func<TResult>** : Accepts zero parameters and returns a value of the type specified by the TResult parameter
- **Func<T, TResult>** : Accepts one parameter and returns a value of the type specified by the TResult parameter
- **Func<T1, T2, TResult>** : Accepts two parameters and returns a value of the type specified by the TResult parameter

Example: `Func<int, string, bool>` accepts two parameters of type **int** and **string** and returns a value of type **bool**.



QA

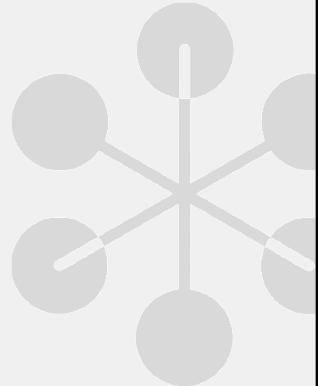
The action delegate

Action<> is a generic delegate which accepts zero or more input parameters and does **not** return a value.

There are many overloads:

- **Action** : Accepts zero parameters and does not return a value
- **Action<T>** : Accepts one parameter and does not return a value
- **Action<T1, T2>** : Accepts two parameters and does not return a value

Example: **Action<int, string, bool>** accepts three parameters of type **int**, **string**, and **bool** and does not return a value.



QA

Func example

The **Add** method accepts two **int** parameters and has a return type of **int**.

It is an instance method (not static).

```
public class DelegateExamples
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

You can instantiate an object instance and use the **Func** generic delegate to encapsulate the **Add** method.

You invoke the **Add** method by invoking the **delegate** and passing the required parameters.

```
DelegateExamples examples = new();
Func<int, int, int> funcAdd = examples.Add;
Console.WriteLine(funcAdd(20, 40));
Console.WriteLine(funcAdd(2, 5));
```

QA

Action Example

Delegates can encapsulate static or instance methods:

```
public class DelegateExamples
{
    1 reference
    public void DisplayGreeting(string name)
    {
        Console.WriteLine($"Hello {name}");
    }

    1 reference
    public static void DisplayGreetingStatic(string name)
    {
        Console.WriteLine($"Hello {name}");
    }
}
```

```
DelegateExamples examples = new();

Action<string> funcHello = examples.DisplayGreeting;
funcHello("Everyone");
funcHello("World");

Action<string> funcHelloStatic = DelegateExamples.DisplayGreetingStatic;
funcHelloStatic("Everyone");
funcHelloStatic("World");
```

QA

Example Part 1

```
public class Book
{
    3 references
    public Book(string title, decimal price, int yearPublished)
    {
        Title = title;
        Price = price;
        YearPublished = yearPublished;
    }
    6 references
    public string Title { get; set; }
    2 references
    public decimal Price { get; }
    2 references
    public int YearPublished { get; }
```

```
static List<Book> FindCheapBooks(List<Book> books)
{
    List<Book> output = new List<Book>();
    foreach (Book book in books)
    {
        if (book.Price < 15M)
        {
            output.Add(book);
        }
    }
    return output;
}
```

```
static List<Book> FindRecentBooks(List<Book> books)
{
    List<Book> output = new List<Book>();
    foreach (Book book in books)
    {
        if (book.YearPublished > 2000)
        {
            output.Add(book);
        }
    }
    return output;
}
```

The two methods are almost identical and only differ by the conditional test

```
List<Book> books = new List<Book>()
{
    new Book("Gulliver's Travels", 10M, 1726),
    new Book("War and Peace", 20M, 1869),
    new Book("This is going to hurt", 5M, 2017),
};

List<Book> cheapBooks = FindCheapBooks(books);
List<Book> recentBooks = FindRecentBooks(books);
```

QA

Example Part 2

```
static List<Book> FindBooks(List<Book> books, Func<Book, bool> func)
{
    List<Book> output = new List<Book>();
    foreach (Book book in books)
    {
        if (func(book))
        {
            output.Add(book);
        }
    }
    return output;
}
```

The two methods are consolidated into one and accept a **Func** delegate which can be used to encapsulate a matching method which contains the conditional test.

```
// these methods accept a Book parameter and return a bool
static bool CheapBook(Book book)
{
    return book.Price < 15M;
}
static bool RecentBook(Book book)
{
    return book.YearPublished > 2000;
}
```

These two methods match the **Func** delegate signature and contain the conditional tests.

```
List<Book> books = new List<Book>()
{
    new Book("Gulliver's Travels", 10M, 1726),
    new Book("War and Peace", 20M, 1869),
    new Book("This is going to hurt", 5M, 2017),
};

// Call FindBooks passing a method that matches the Func<Book, bool> signature
List<Book> cheapBooks = FindBooks(books, CheapBook);
List<Book> recentBooks = FindBooks(books, RecentBook);
```

QA

Lambdas

- A **lambda** expression is used to create an anonymous function
- You use the lambda declaration operator `=>` to separate the lambda's parameter list from its body
- A lambda expression can be either an expression lambda (single line) or a statement lambda (multiple lines enclosed in braces)
- Specify input parameters on the left side of the lambda operator or use empty brackets if there are zero parameters
- If there is only one parameter, brackets are optional
- Any lambda expression can be converted to a delegate type

```
Action line = () => Console.WriteLine(); // zero parameters
Func<double, double> cube = x => x * x * x; // one parameter x
Func<int, int, bool> testForEquality = (x, y) => x == y; // two parameters x and y
```



QA

Func example as a lambda

This example uses a **delegate** that encapsulates a *named* Add method.

```
public class DelegateExamples
{
    1 reference
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

```
DelegateExamples examples = new();
Func<int, int, int> funcAdd = examples.Add;
Console.WriteLine(funcAdd(20, 40));
Console.WriteLine(funcAdd(2, 5));
```

This example uses a **lambda** expression to encapsulate an *anonymous* method.

```
Func<int, int, int> add = (x, y) => x + y;
Console.WriteLine(add(20, 40));
Console.WriteLine(add(2, 5));
```

QA

Action Example as a lambda

These examples use **delegates** that encapsulate *named* methods:

```
1 reference
```

```
public class DelegateExamples
{
    1 reference
    public void DisplayGreeting(string name)
    {
        Console.WriteLine($"Hello {name}");
    }

    1 reference
    public static void DisplayGreetingStatic(string name)
    {
        Console.WriteLine($"Hello {name}");
    }
}
```

```
DelegateExamples examples = new();
```

```
Action<string> funcHello = examples.DisplayGreeting;
funcHello("Everyone");
funcHello("World");
```

```
Action<string> funcHelloStatic = DelegateExamples.DisplayGreetingStatic;
funcHelloStatic("Everyone");
funcHelloStatic("World");
```

These examples use **lambda** expressions to encapsulate *anonymous* methods:

```
Action<string> greet = (string name) => Console.WriteLine($"Hello {name}");
greet("Everyone");

Action<string> greet2 = name => Console.WriteLine($"Hello {name}");
greet2("World");
```

QA

Example Part 3

```
static List<Book> FindBooks(List<Book> books, Func<Book, bool> func)
{
    List<Book> output = new List<Book>();
    foreach (Book book in books)
    {
        if (func(book))
        {
            output.Add(book);
        }
    }
    return output;
}
```

The two methods are consolidated into one and accept a **Func** delegate, which can be used to encapsulate a matching method which contains the conditional test.

Named methods are no longer required since the conditional tests are now defined as **lambda** expressions which match the `Func<Book, bool>` delegate signature.

```
List<Book> books = new List<Book>()
{
    new Book("Gulliver's Travels", 10M, 1726),
    new Book("War and Peace", 20M, 1869),
    new Book("This is going to hurt", 5M, 2017),
};

// Call FindBooks passing a lambda expression that matches the Func<Book, bool> signature
List<Book> cheapBooks = FindBooks(books, b => b.Price < 15M);
List<Book> recentBooks = FindBooks(books, b => b.YearPublished > 2000);
```

QA

The Predicate delegate

Predicate<> is a generic delegate which accepts one parameter and a return type of **bool**.

Example: **Predicate<int>** accepts one parameter of type **int** and returns a value of type **bool**.

```
Predicate<int> oldEnough = a => a >= 21;  
Console.WriteLine(oldEnough(22)); //True  
Console.WriteLine(oldEnough(18)); //False  
  
Predicate<Book> isCheapBook = b => b.Price <= 5M;  
Book book = books[2];  
string isCheap = isCheapBook(book) ? " is " : " is not ";  
Console.WriteLine(book.Title + isCheap + " a cheap book");  
// This is going to hurt is a cheap book
```

QA

You may come across a built-in delegate called **Predicate**.

Func<T, bool> is the equivalent of the **Predicate** delegate.

For example:

Func<int, bool> is equivalent to **Predicate<int>**.

Func<string, bool> is equivalent to **Predicate<string>**.

Because **Func** can do everything **Predicate** can do it is recommended that **Func** should be used instead of **Predicate**.

Delegate/ lambda examples

```
// Arrays and Lists have many methods that use a Predicate delegate
// Find uses a Predicate delegate
Book? recentBook = books.Find(book => book.YearPublished >= 2015);
if (recentBook != null)
{
    Console.WriteLine(recentBook);
}

// FindAll uses a Predicate delegate
List<Book> startsWithW = books.FindAll(book => book.Title.StartsWith("W"));

// ForEach uses an Action delegate
startsWithW.ForEach(book => global::System.Console.WriteLine(book.Title));
//output: War and Peace

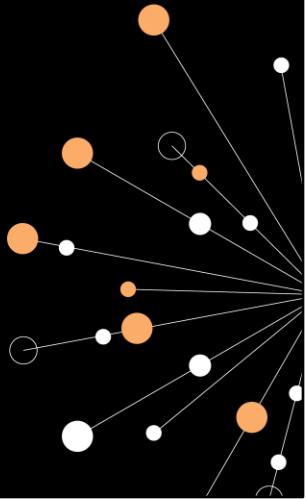
// Average uses a Func<T, decimal> delegate
decimal averagePrice = books.Average(b => b.Price);
Console.WriteLine(averagePrice);
//output: 11.66666667
```

QA

Summary

- Delegates
- The Func delegate
- The Action delegate
- Lambda expressions
- The Predicate delegate
- Delegate and Lambda examples

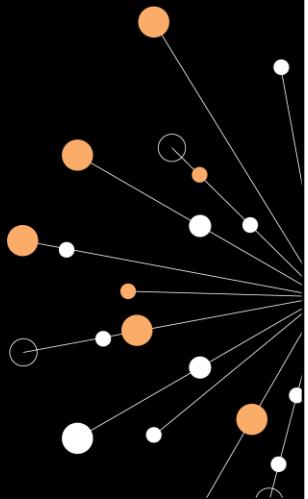
QA



Hands on Lab

Exercise 08 Delegates and Lambdas

QA



09 Exception Handling

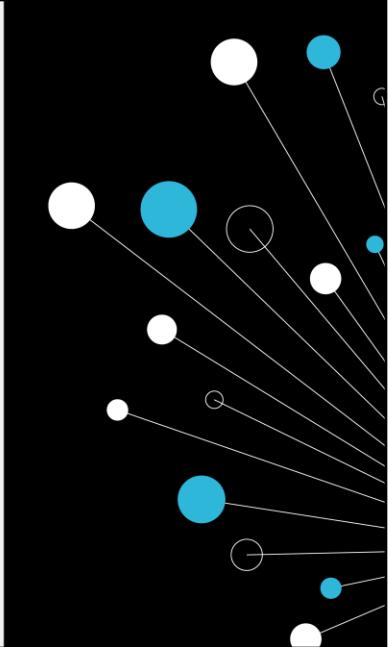
C# for HAAS F1



Learning objectives

- Exception handling
- Example: Try, catch, finally
- Understanding execution flow
- Throwing exceptions
- Custom exceptions
- Filtered exceptions
- Inner exceptions
- Best practices

QA

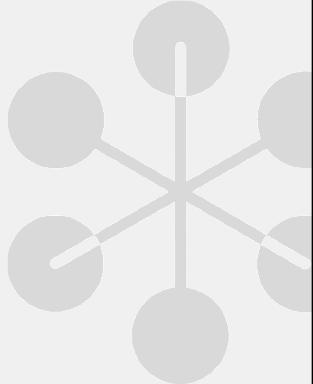


Exception Handling

The **exception handling** features of the C# language let you deal with any unexpected or exceptional situations that occur whilst your code is running

There are four keywords used:

- **Try**: Try actions that may not succeed
- **Catch**: Handle failures
- **Finally**: Clean up resources
- **Throw**: Generate an exception



QA

Programs will fail from time to time, particularly when accessing resources in other processes or across a network: printers can run out of paper, databases can become unavailable and users can enter incorrect security credentials.

The .NET Framework takes a single approach to error reporting: the throwing and catching of exceptions.

Exception handling is designed to ensure that programs take note of problems as they occur, thus hopefully stopping them writing garbage data or causing a traumatic system failure.

Exceptions are types. They have properties and methods.

Example: Try Catch finally

```
SqlConnection conn = new SqlConnection();
try
{
    conn.Open();
    //do things with connection
}
catch (SqlException ex)
{
    Console.WriteLine("Data access error: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("General error: " + ex.Message);
}
finally
{
    if (conn != null && conn.State == ConnectionState.Open)
    {
        conn.Close();
    }
}

//remainder of method
```

QA

The three components of exception handling code are the try, catch and finally blocks. These work as follows:

try

The try block is where we write the code that is the normal flow of execution for the application. In this example this consists of opening a connection to a SQL Server database, and performing operations against it (code not shown).

catch

Following a try block there can be zero-to-many catch blocks. These contain statements that are to be executed if an exception of the specified type is thrown (raised) by the code that is called in the try block. If there is more than one catch block they must be in a very specific order: the most derived exception type blocks must appear before their base class counterparts. This is because the exception handling code walks the blocks in sequence, looking for the first exception that matches according to the "is" operator. Only ONE of the exception handling blocks will ever be executed.

finally

Optionally, there can be a finally block. Unless there are no catch blocks, in which case it is required. The code in this block will always execute, no matter whether an exception is thrown (and/or caught) or not.

A try block must be followed EITHER by one or more catch blocks and optionally a finally block OR by a finally block.

The next slides illustrate how this fits together and how exceptions affect the flow of execution of code.

Understanding Execution Flow: 1

```
public class Program {
1  static void Main() {
2    try {
3      Task.F1( 0 );
4      Task.F2();
5    }
6    catch (Exception ex)
7    {
8      Console.WriteLine(ex.Message);
9    }
10}
11}
12}

public class Task {
13  public static void F1(int a) {
14    F3(a);
15    F4();
16  }
17
18  public static void F2() { }

19  public static void F3(int y) {
20    int x = 10 / y;
21    // Does not run
22  }
23  public static void F4() { }
24}
```

Step

QA

In this first example, we have a simple program that, under normal conditions, would execute two methods (Task.F1 and Task.F2). The method F1 calls the methods F3 and F4 respectively.

However, there is a potential problem in F3. When we divide by zero using integer division the result is undefined. Consequently, the author of the System.Int32 class decided that they would throw an exception of type **System.DivideByZeroException** under these circumstances.

When this exception is thrown, the CLR starts to "unwind the stack" until it finds a matching exception handler for **DivideByZeroException** (or one of its base classes). Therefore, the code after the divide by zero action in the method F3 doesn't get executed. Since the call to F3 is not inside a try {} block there can be no catch handler here. Therefore, the stack continues to unwind and F4 is not executed.

The stack has now unwound to the point that F1 was invoked (in Main). This call is inside a try{} block, so the exception handling code looks to see if there is a matching catch{} block. It locates the block for **System.Exception**, which is the ultimate base class for all exception types, so the thrown exception matches the "is a type of" test. Control is therefore transferred to this catch{} block and the stack unwinding process is now complete. The catch{} block simply displays the exception's error message information on the console. The code following the catch{} block then executes, which means that Main returns as normal. Therefore, F2 does not get executed.

Understanding Execution Flow: 2

```
public class Program {
1 static void Main() {
2     try {
3         Task.F1( 0 );
4         Task.F2();
5     }
6     catch (Exception ex) {
7         Console.WriteLine(
8             ex.Message);
9     }
10}
11}
```

Step

QA

```
public class Task {
11    public static void F1(int a) {
12        F3(a);
13        F4();
14    }
15
16    public static void F2() { }
17
18    public static void F3(int y) {
19        int x;
20        try {
21            x = 10 / y;
22            // Does not run
23        }
24        catch (DivideByZeroException ex)
25        { }
26        // Rest of method
27    }
28
29    public static void F4() { }
30}
```

In this example, the developer of the Task class has anticipated that someone might call the method F3 and pass in the value zero. They have therefore protected the division operation with a **try{} catch{}** block to catch any divide by zero exceptions. Here's the flow of execution when 0 is passed in:

Main calls Task.F1, passing in zero.

Task.F1 calls Task.F3, passing the value 0 through in the 'a' parameter.

Task.F3 attempts to perform its division operation, and a

DivideByZeroException is thrown by the internals of the System.Int32 class.

The stack begins to unwind. As the code in F3 is inside a **try{}** block, the CLR looks to see if there is a matching **catch{}** block. There is, which means that control is passed to the **catch{}** block. The code after the division by zero operation in F3 doesn't execute.

As the exception has been caught in F3, the method F3 returns as normal to F1. Method F4 then executes as normal, and assuming that no exception is generated, control is returned to F1 as normal. F1 then returns to Main, and F2 executes. Again, assuming that no exception is thrown in F2, control returns to Main as normal. As no exception was thrown, without being caught, by the methods F1 and F2 (and the methods they called in turn) the **catch{}** block is skipped, and Main returns as normal.

Understanding Execution Flow: 3

```
public class Program {
1  static void Main() {
2    try {
3      Task.F1( 0 );
4      Task.F2();
5    }
6    catch (Exception exn)
7    {
8      Console.WriteLine(
9        exn.Message);
10    }
11}
```

Step

QA

```
public class Task {
11  public static void F1(int a) {
12    F3(a);
13    F4();
14  }

15  public static void F2() { }

16  public static void F3(int y) {
17    int x;
18    try {
19      x = 10 / y;
20      Console.WriteLine(x);
21    }
22    finally {
23      Console.WriteLine("Other");
24    }
25    // does not run if try fails
26  }

27  public static void F4() { }
28}
```

This example highlights the use of **finally** to ensure the execution of code, no matter whether an exception is thrown or not. As ever, Main calls Task.F1, passing in the value zero. F1 calls F3, passing the value zero through. F3 attempts to perform the division, but the System.Int32 class throws the **DivideByZeroException**. The CLR starts to unwind the stack and locates the **finally{} block** that follows the **try{} block** in which the division was performed. It therefore executes the code inside the **finally** block, and displays “Other” to the console.

At this point the CLR is still looking for a **catch{}** block to handle the exception; remember the **finally{}** block only guarantees the execution of code and doesn't handle exceptions. The stack is therefore unwound until the **catch{}** block is located in Main, where the exception's message is written to the console. In this example, the code that writes the value of ‘x’ to the console isn't executed if an exception is thrown by the division operation (nor is the code in F4 or F2). You can therefore see the benefit of the **try / finally** construct when you need to ensure that certain code is always executed. This is particularly important for ensuring the database connections and file streams are always closed.

Throwing Exceptions

To generate an exception, you throw a reference to an exception object:

```
void PrintReport(Report rpt) {  
    if (rpt == null) {  
        throw new ArgumentNullException  
            ("rpt", "Can't print a null report");  
    }  
    ...  
}
```

You can re-throw a caught exception which maintains the original stack trace:

```
catch (ArgumentNullException ex) {  
    ...  
    throw;  
}
```

QA

There will be times when we will want to throw exceptions. For example, when writing a method that takes an object reference as a parameter. Our code might reasonably expect this reference to be set to a valid object, rather than be null. Consequently, we can test it to see whether it is null and if it is we can then throw a **System.ArgumentNullException**.

Most exceptions provide a set of overloaded constructors through which we can provide additional information. In the example above, the exception lets us specify the parameter name and a helpful message.

We can also re-throw exceptions that we have caught. This allows us to perform some clean up code in response to the exception but then propagate the exception back up the call stack. To re-throw the exception simply use the “**throw**” keyword by itself inside a **catch** block.

This type of throw statement preserves the original exception, including the originating stack trace.

It should be noted that many of the exception classes also support overloaded constructors that accept an “inner exception”. An example of when this might be of use is in a Data Access Layer component (one that talks to a database), where we want to enclose the low level exception (to preserve traceability) inside a custom exception that we are developing (to provide a domain-specific exception type).

Custom Exceptions

Derive the class from **System.Exception**

```
public class CarFactoryException : Exception
{
    // Other overloaded constructors / properties / fields
}

1 reference
public class InvalidModelError : CarFactoryException
{
    public InvalidModelError()
    {
    }
}
```

To provide rich exception details:

- Overload the constructor to pass in information
- Provide public properties to allow retrieval

QA

We can create custom domain-specific exception classes for use in our applications. All exception types are derived from **System.Exception**. Custom exception types are no different in this respect.

Don't create too many exception types, and never duplicate exceptions that exist in the Framework Class Library. In most cases, one of the predefined exceptions is perfectly adequate. An excess of exceptions can make it difficult to write readable code.

Filtered Exceptions

- **User-filtered exception handlers** catch and handle exceptions based on requirements you define
- Use the **when** keyword with the **catch** statement:

```
try
{
    throw new MyException() { MinorFault = true };
}
catch (MyException ex)
{
    if (!ex.MinorFault)
    {
        throw;
    }
    Console.WriteLine("deal with minor fault");
}

try
{
    throw new MyException() { MinorFault = false };
}
catch (MyException ex) when (ex.MinorFault)
{
    Console.WriteLine("deal with minor fault");
}
catch (MyException ex) when (ex.MinorFault == false)
{
    Console.WriteLine("deal with major fault");
    throw;
}
```

QA

Inner Exceptions

The **Exception** class defines an **InnerException** property that enables you to wrap a custom exception around a system exception, whilst maintaining traceability as to the original cause of the exception.

Your custom Exception type requires a constructor that accepts an inner exception

The **Message** and **InnerException** properties are read-only so call the base class constructor to set their values

```
public class InvalidPaintJobException : Exception
{
    // ...
    public InvalidPaintJobException()
    {
    }

    public InvalidPaintJobException(string message, Exception inner) : base(message, inner)
    {
    }
}
```

```
SqlConnection conn = new SqlConnection();
try
{
    conn.Open();
    //look up the required paint colour
}
catch (SqlException ex)
{
    throw new InvalidPaintJobException(
        message: "not a valid colour spec",
        inner: ex);
}
```

QA

There are a few .NET framework exceptions that will have an inner exception populated, notably the **HttpUnhandledException** in ASP.NET; by the time the server sees an unhandled exception, it has been “wrapped” in this general purpose object, whose **InnerException** property will contain the exception that was originally unhandled.

Use InnerExceptions with care. Your client code should not have to dig down more than one or two levels to find the original exception.

Imagine needing the following code:

```
Exception eek =
ex.InnerException.InnerException.InnerException.InnerException;
```

There is a **GetBaseException** method on the **Exception** type which will find the original exception (the one furthest down the chain whose **InnerException** is null).

Best Practice

- Use specific catch blocks for the exceptions you expect within the code
- Include a last catch block to catch **System.Exception** which will catch unexpected exceptions
- Not every method needs **try** and **catch** blocks since exceptions are propagated up the call stack
- Use **finally** blocks to tidy up resources
- Only **throw** exceptions if the situation is exceptional rather than expected
- Ensure your tests check that exceptions are thrown when expected
- Do not disclose sensitive or too much information in error messages



QA

Firstly, we shouldn't simply write catch handlers for **System.Exception**. This would catch every type of exception, including those from the CLR that are indicating traumatic problems (stack overflow, security exceptions, etc.). This leads neatly onto the second point: only catch the specific exceptions that are expected. Code which attempts to catch every possible exception type is impossible to read.

Not every method needs **try / catch** blocks. Exceptions propagate back up the call stack, so only place them where, for example, logging or resource management code is required.

To manage key resources such as database connections and files, which need to be closed when finished with, use **try / finally** blocks.

Next, exceptions are designed to be used in exceptional circumstances such as attempting to open a file that doesn't exist, not to indicate an end-of-file (which is expected when reading from them).

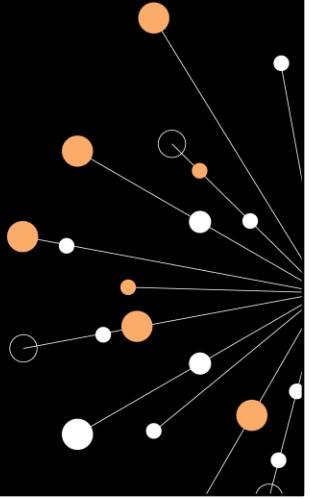
Many developers dutifully add exception handling code and then never test their program under the circumstances where it is called.

Never just display an exception's **Message**. **SqlExceptions** often contain Messages like "table Persons not found". This doesn't seem like much, but a determined malicious user can use these kind of error messages to build up a picture of the system (known as a jigsaw attack). A message saying "invalid password" tells them they've guessed a username correctly.

Summary

- Exception handling
- Example: Try, catch, finally
- Understanding execution flow
- Throwing exceptions
- Custom exceptions
- Filtered exceptions
- Inner exceptions
- Best practices

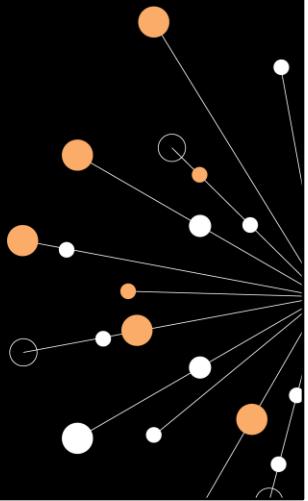
QA



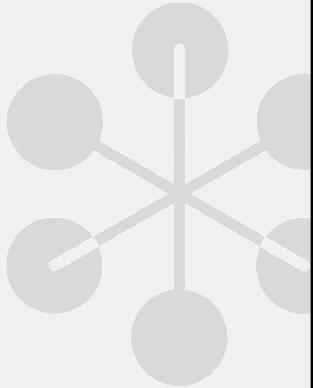
Hands on Lab

Exercise 09 Exception Handling

QA



Appendix



QA

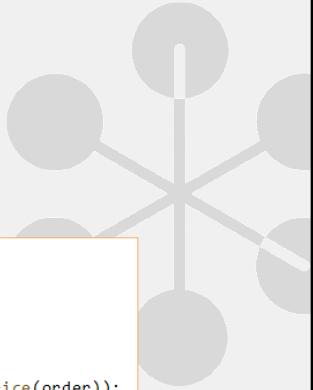
xUnit and Exceptions

- xUnit enables you to test when an exception is thrown specifically within the *Act* stage of your test, as opposed to the *Arrange* or *Assert* stage
- Use `Assert.Throws<TException>` passing a lambda statement to perform the action you are testing
- `Assert.Throws` returns the exception so you can access any properties and make further assertions on those

```
[Fact]
public void Total_Price_Never_Negative()
{
    Checkout checkout = new Checkout(new TestDiscount());
    order.Add(new Pizza(Size.Small_10, Crust.Regular_2));

    Assert.Throws<NegativePriceException>(() => checkout.GetBestPrice(order));
}
```

QA



361

10 Introduction to ASP.NET Core Web API Core MVC

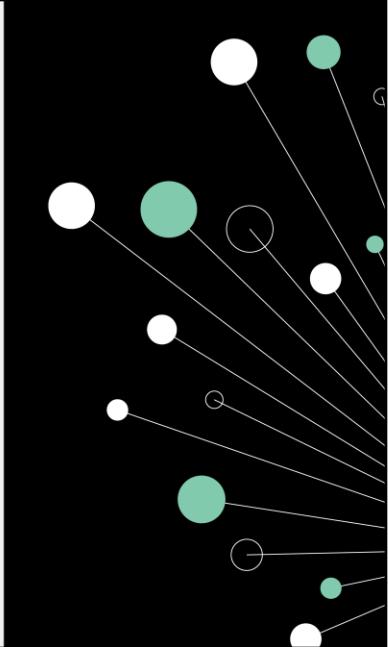
C# for HAAS F1



Learning objectives

Gain an overview of an MVC application

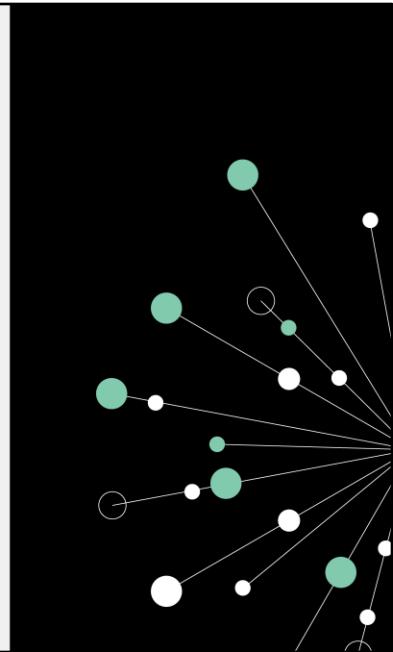
QA



Session Content

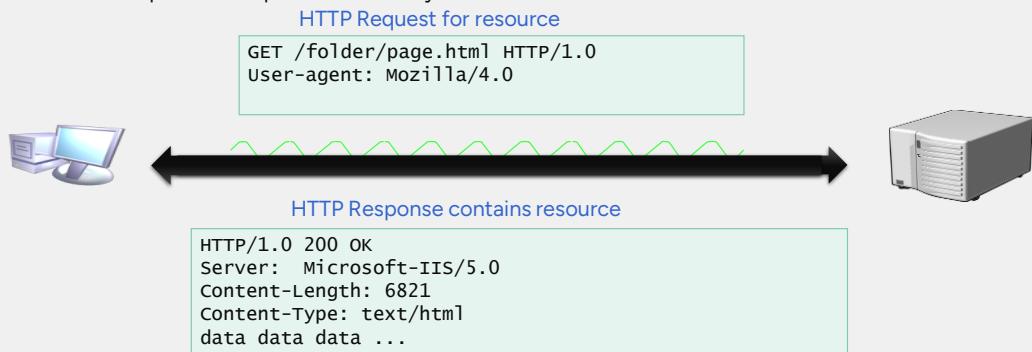
- HTTP
- The MVC Pattern
- Models, Views, and Controllers
- ASP.NET MVC Conventions
- MVC Project structure
- Visual Studio support

QA



Hypertext Transfer Protocol (HTTP)

- HTTP is an Application Level Protocol
 - A request and response protocol
 - Each request is independent from any other



HTTP is stateless. This means that each request for a resource from the Web server, such as that for an HTML page, is seen as being completely independent of any previous request.

The Web server doesn't store any information that associates the different requests, which in turn means that the Web server can scale well to deliver extremely large numbers of requests.

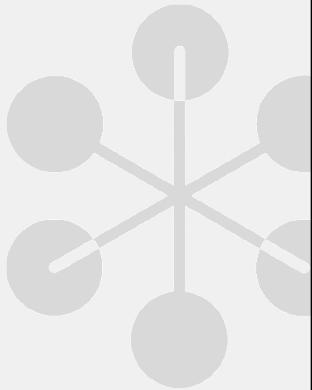
HTTP follows a simple request/response paradigm:

1. The connection is opened.
2. The browser sends an HTTP request for a resource.
3. The server sends an HTTP response containing the resource.
4. The connection is closed.

In HTTP version 1.1, the browser can request that the server keep the connection alive for a short while so that the browser can request another resource without opening another connection to the server.

Some of the advantages of ASP.NET (Core)

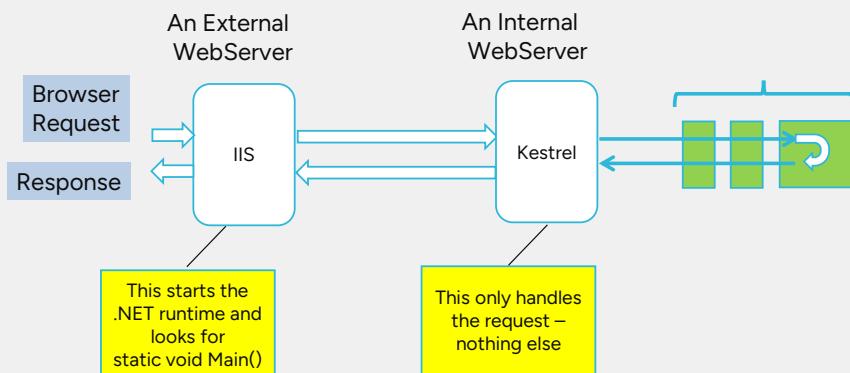
- It is no longer tied to IIS and Windows
- It has a modular request pipeline – so is typically very lightweight
- Is fully open-source and cross-platform
- It is extremely fast



QA

366

The Pipeline



QA

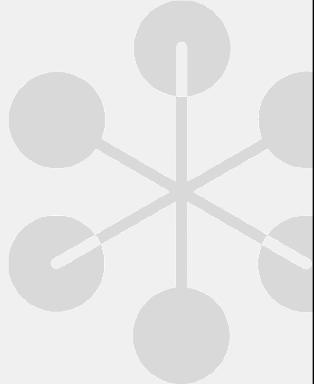
367

What is the MVC Design Pattern

An architectural pattern that can be used when developing interactive applications

- Uses the principle of "Separation of Concerns"
 - Enhancing maintainability, extensibility, and testability
- Separates interaction logic into three areas:
 - Controller
 - Orchestrates the model and the view to produce the desired response
 - Model
 - Responsible for producing the data just for a view
 - View (not relevant in API setting)
 - Responsible for rendering the response
- Also:
 - Routing
 - Directs the user request to the correct controller and action

QA

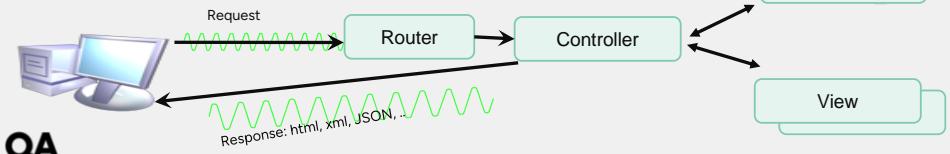


Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts.

- A controller sends commands to the model to update and retrieve the model's state. It also selects the associated view to be returned.
- A model represents the information in the system. That information could be stored in a database, a file, or in memory. Its underlying store is abstracted way by the model, which provides a single interface upon which to interact with the data.
- A view renders the information from the model, it uses to generate the response to the user. In a RESTful API environment there is little or no need to include View functionality

MVC Architecture

- Controller
 - Translates user actions (http requests, Ajax calls, etc.) into application function calls, Works on the model, and selects the appropriate View based on preferences and Model state.
- Model
 - Probably better thought of as a ViewModel – ie. the extract of the business data specific for a view.
- View (**Not relevant in an API setting**)
 - Renders the content of the response (in Html).
 - CSS is used to turn this content into presentation



Routing

Traditionally URLs map to file names

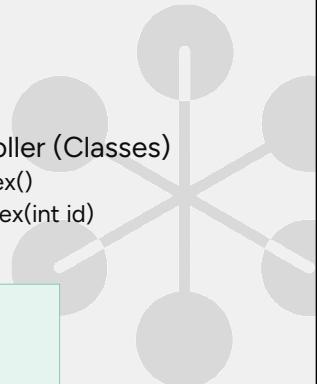
- <http://www.mysite.com/default.html>
- <http://www.mysite.com/Home/details.aspx>

In MVC URLs are mapped to Actions (Methods) inside Controller (Classes)

- <http://www.mysite.com/Home/Index> => HomeController.Index()
- <http://www.mysite.com/Home/Index/5> => HomeController.Index(int id)
- In REST API applications this is generally done via "attribute routing":

```
[HttpGet("")]
[HttpGet("Home")]
[HttpGet("Home/Index")]
[HttpGet("Home/Index/{id?}")]
public IActionResult Index(int? id) {
    return GetMessage(id);
}
```

QA



Before ASP.NET MVC, URLs in web application mapped to physical files at a disk location. So, for example if you had a URL

'<http://www.mysite.com/students/list.aspx>' it simply meant there was a list.aspx file in a 'students' folder. The URL had no other meaning. However, if this was an MVC site, the URL could look like '<http://www.mysite.com/student/>'. By convention the URL would map to the StudentController class with an action named Index.

The default route takes the first part of a Url path and assumes it contains the name of the controller, it assumes the second part of the URL, if it exists, is the name of an action in that controller. Allowing it to create the controller and invoke the action method inside that controller.

Model

We're going to need some data (in the Models folder):

```
public class Forum
{
    public int ForumId { get;set; }
    public string Title { get;set; }
```

Note how we've added an Id – because we intend this to be stored in a database

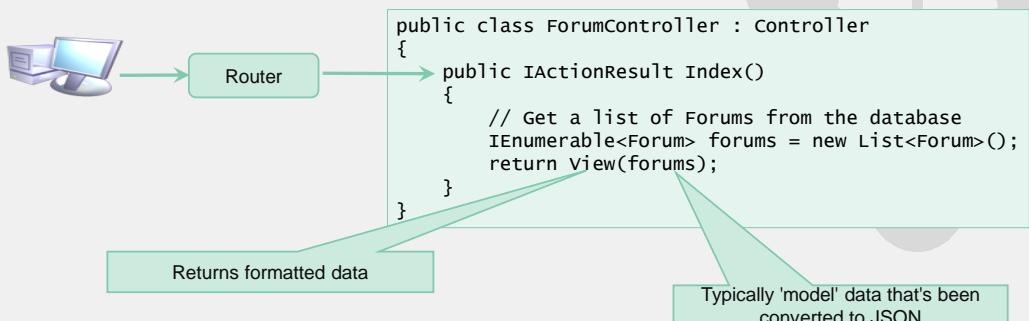


QA

Controller

Controller

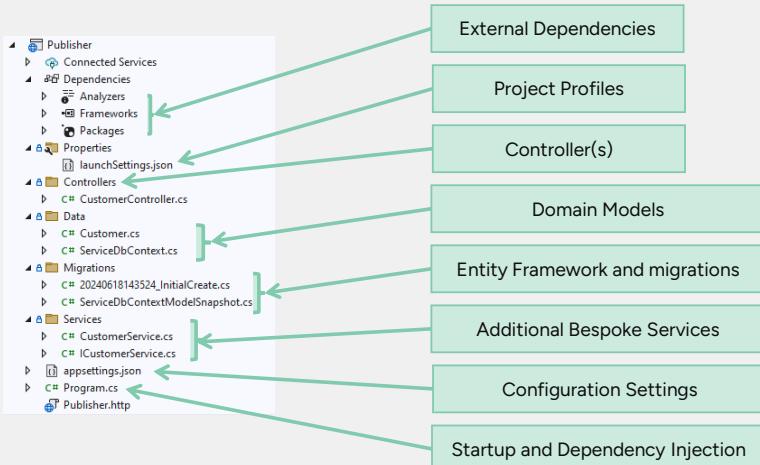
- Created and invoked by the Router when an HttpRequest is received
- Methods inside a controller are referred to as **Actions**



QA

The MVC controller is just a class that has inherited from the Controller base class. You add methods to this controller class to implement the functionality of the application. Methods are invoked by requests sent to the controller and are referred to as Actions. It is these Actions that control the interaction between the user and the application. A typical Action would create an appropriate model and convert the model to JSON before returning it.

Important folders/files in your application



QA

ASP.NET MVC API projects are very “Conventional”.

Visual Studio assumes that all controllers are post fixed with Controller. E.g. `HomeController`. Therefore, any routes such as `/Home/Index` will be routed to the controller named `HomeController`.

It assumes that all views are located in a folder named after their controller. E.g. Views belonging to `HomeController` would be located in `Views\Home`. Views are assumed to be named after the action that returns them. E.g. An action named `Create` would return a view named `Create`.

Hands on Lab

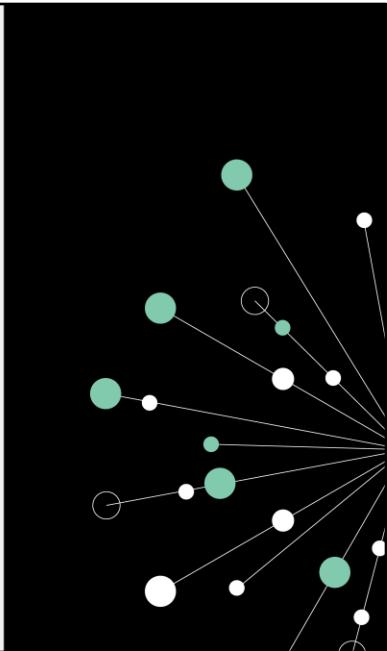
10 Introduction to ASP.NET MVC API Core

Objective:

Your goal is to make sense of the fundamentals of ASP.NET MVC API Core

You start out by creating a basic ASP.NET MVC API Core project and then explore how to go about adding a controller and giving it a set of Actions. You will explore how C# method overloading causes issues that can be overcome by adding routing via `HttpGet` attributes. You will test the applications by using the built in Swagger capabilities and also take a look at using Postman as an alternative.

QA

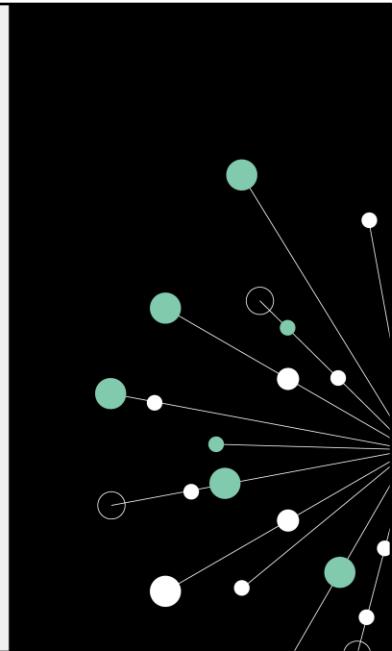


Review

Gain an overview of an MVC application

- The MVC Pattern
- Routers, Controllers, Models, View-Models, Views and Areas
- MVC Project structure
- Visual Studio support

QA



Appendix: Description of Important Folders in your Application

Folder	Description
App_Data	Contains application data files including MDF files, XML files as well as other data store files. The App_Data folder is used by ASP.NET MVC to store an application's local database, which can be used for maintaining membership and role information.
Content	Contains .css files as well as image files and generic resources that define the appearance of views.
Controllers	This folder contains the controller class files.
Models	We rename this to BuiltinModels as it contains the Model and ViewModel classes required by the built-in authentication mechanism.
Scripts	This folder contains script files such as those required for jQuery or Ajax.
Views	This folder contains one folder for each controller declared in the Controllers folder. These folders contain the view files (.aspx or .cs/vb]html)
Views\Shared	This folder contains resources required by all views. E.g. _Layout.[cs/vb]html

QA

11 Unit Testing

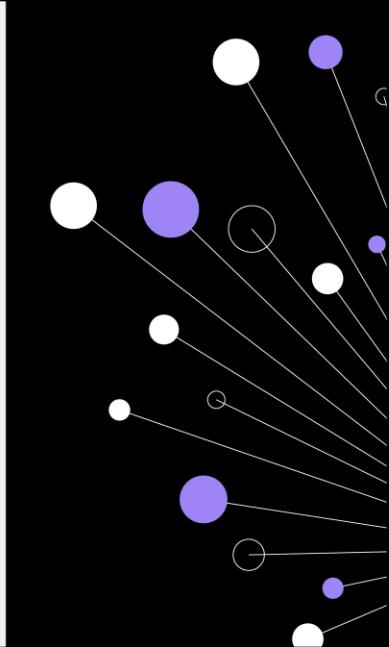
C# for HAAS F1



Learning objectives

- What is the purpose of Unit Testing?
- The green-red cycle
- Refactoring is essential
- Mocking

QA



Traditional Development Cycle

Problem: High level of defects

- Lengthy testing phase after a release is frozen
- Cost of fixing bugs is far higher than if bugs are caught when introduced into code

Problem: Poor maintainability

- Legacy spaghetti code that "works"
- Can't be touched – fear of breaking



QA

Testing is not optional

s/w *always* gets tested

If not by you, then by the customer/user

- This may not be for some time after deployment

The longer the time delay between writing the bug and finding it, the more expensive it is to fix.

- Actually, it is not just more expensive, it is much, much more expensive.



QA

Test-Driven Cycle

Note the process:

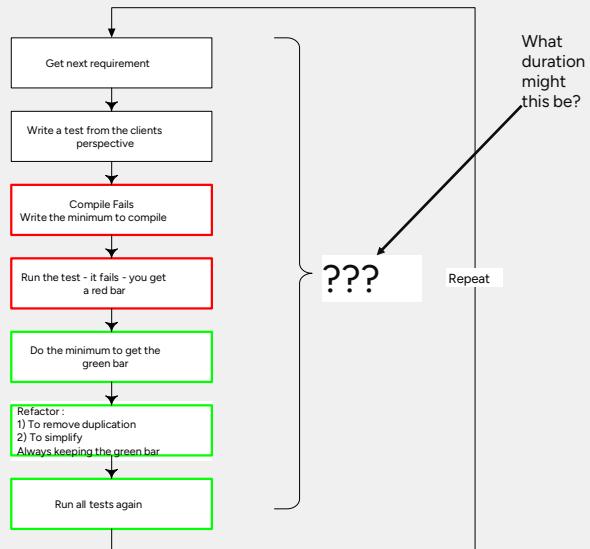
1) Write a test

2) Write the code

3) Run the test

4) Refactor

QA



The mantra of the unit test approach is that you only write operational code under one of 2 circumstances

You have written a test and you have not yet written the operational code to allow the test to even compile

b) The test compiles but does not work.

You write the absolute minimum code to allow the test to work.

It is entirely normal that at some later stage you return to the operational code and refactor it. Clearly after any refactor the tests must still pass.

This brings us to the second unit test mantra

You either write new functionality or refactor existing functionality. You never do both simultaneously.

The Unit Test Development Mindset

Stay on Green – if on Red stop all else & move to Green

There are only 3 conditions for writing operational code

1. Test doesn't compile
2. Test fails
3. Refactor behind existing tests

Only test the s/w through checked-in unit tests

Refactor relentlessly

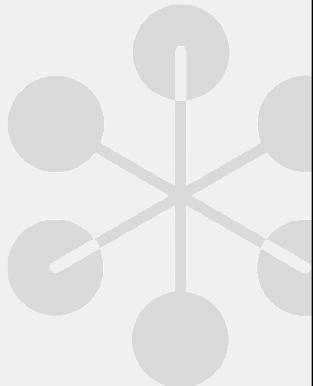
Do all this in small iterations



QA

The benefits are really for the developer

- You fix all the trivial problems as you go along
- You know that they have not recurred
- You document, without effort, how you see other s/w interfacing with your s/w
- You are able to refactor your code to make it more maintainable, faster, ... knowing that you haven't broken anything.



QA

Unit Testing Best Practices

Test naming

- Document test methods by a suitably descriptive name eg
- What_You_Are_Testing_AND_Expected_Outcome

Make unit tests as fine-grained as possible

- Test one object at a time
- One of the main purposes of a Unit Test is to quickly identify the reason for a failure – that's why we keep them small

Don't put more than one test into one TestMethod()

- The more complex a test becomes, the greater the risk that the test itself will need debugging

Keep tests independent of each other

- Implies tests are order-independent

Think AAA – see next slide



QA

AAA

You should get in the habit of partitioning your test methods by:

```
//Arrange  
//Act  
//Assert
```

Arrange: Data for the Code-Under-Test is specified here along with the expected result(s)

Act: The Code-Under-Test is called.

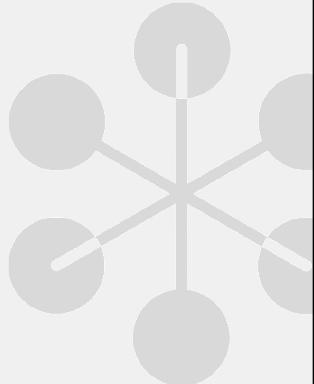
Assert – Use Assert methods to ensure result(s) is as expected



QA

Mocking Dependency Behaviour with FakeltEasy

- FakeltEasy is a .NET library for creating fake (mock) objects
- Used in unit testing to mock the behaviour of dependencies.
 - No need to use genuine objects that may run slowly or lack versatility
- Allows components to be tested in isolation
- FakeltEasy is:
 - Relatively easy to use
 - only one of a number of mocking technologies



QA

Mocking technologies are extremely useful and are used to simplify the testing of isolated components.

FakeltEasy is a lightweight and user-friendly library for mocking dependencies in unit tests. It is designed to work seamlessly with modern .NET testing frameworks like xUnit, NUnit, and MSTest. It is capable of faking methods, properties, and even void-returning methods with minimal setup.

Other mocking technologies for C# include:

Why use a Mocking Technology?

- Dependencies can be complex and/or have unpredictable behaviour
- Using a mocking technology ensures tests focus on the Component Under Test (CUT)
- Can be used to fake external systems such as databases and APIs
- Can significantly improve the speed of tests and their reliability



QA

There are some distinct challenges associated with the testing of components that have real dependencies, such as database or network latency, setup complexity, or availability issues. Use of a mocking technology like FakeltEasy helps by creating controlled and predictable test environments thus ensuring a more deterministic testing process.

A Simple Example of FakItEasy

Define an interface for the dependency:

```
Public interface IService {  
    Employee GetEmployee(int id);  
}
```

Mock the dependency:

```
//Arrange  
using FakeItEasy;  
IService service = A.Fake<IService>();  
Employee emp = new Employee(){id=1,...}  
A.CallTo(() => service.GetEmployee(1)).Returns(emp);
```

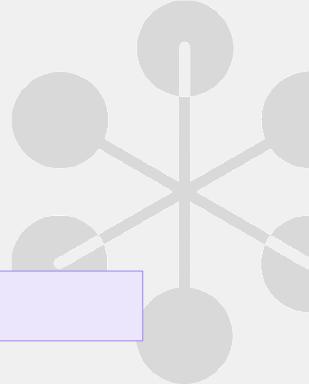
Invoke the Component Under Test

```
//Act  
//Service dependency injected into constructor  
Employee emp1 = component.MethodBeingTested(service);
```

Use the fake in tests

```
// Assert  
A.CallTo(() => service.GetEmployee(1)).MustHaveHappened();  
Assert.Equal(1, emp1.id);
```

QA



Firstly, create a fake instance of the dependency. Use the A.CallTo method sets up expected behaviour for the fake. Note that you can verify interactions (e.g., whether a method was called) using FakItEasy.

Common Usages for Mocks and Fakes

Replacing database access in unit tests

- Database access can be slow
- Updates, Additions and Deletions from a database may break a test that was working
- Mocking APIs
 - Can be slow to respond
 - May not consistently return the same results
- Simulating error scenarios
- Testing component interactions

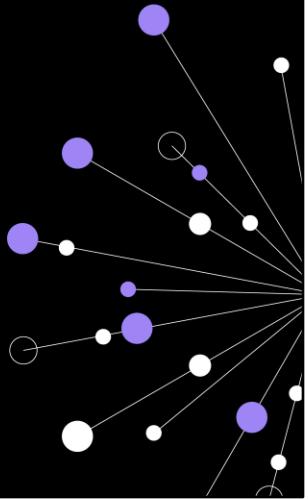


QA

Summary

- What is the purpose of Unit Testing?
- The green-red cycle
- Refactoring is essential
- Mocking

QA



Activity

Exercise 11 Unit Testing

QA

