



**ChairBear** Digital Learning

---

# Software Language Theory

Handbook for Software Development Apprentices

|  |          |
|--|----------|
| <b>Software Language Theory</b>              | <b>1</b> |
| Foreword                                     | 6        |
| The Software Development Lifecycle           | 7        |
| Why Develop New Systems?                     | 7        |
| Feasibility Study                            | 7        |
| Requirements Engineering                     | 8        |
| Requirement Levels                           | 9        |
| Functional Requirements                      | 10       |
| Non-Functional Requirements                  | 10       |
| Functional VS Non-Functional Summary         | 12       |
| Design                                       | 12       |
| The Purpose Of Design                        | 12       |
| Logical                                      | 13       |
| Physical                                     | 13       |
| Algorithms                                   | 13       |
| Pseudo Code                                  | 15       |
| Flowcharts                                   | 15       |
| Types Of Algorithm                           | 17       |
| Categories Of Algorithm                      | 19       |
| Computational Thinking                       | 20       |
| Decomposition (Functional Decomposition)     | 21       |
| Pattern Recognition                          | 21       |
| Abstraction                                  | 21       |
| Design Techniques & Documentation            | 21       |
| Software Architecture                        | 22       |
| Infrastructure Architecture                  | 23       |
| Data Modelling                               | 23       |
| Interface Design                             | 24       |
| Design Principles                            | 24       |
| Data Design                                  | 25       |
| Data Dictionary                              | 25       |
| Entity Relationship Diagram                  | 25       |
| Design Constraints                           | 25       |
| Building In Security                         | 26       |
| Pro-active Security Approaches Summary Table | 27       |
| Common Security Attacks                      | 28       |
| Development                                  | 31       |
| Proactive Security Approaches                | 31       |
| Firewalls                                    | 33       |



|                                    |    |
|------------------------------------|----|
| VPN                                | 33 |
| DMZ                                | 34 |
| Prototyping                        | 34 |
| Testing                            | 35 |
| Unit Testing                       | 35 |
| Component Testing                  | 35 |
| System Testing                     | 35 |
| Acceptance Testing                 | 35 |
| Regression Testing                 | 36 |
| Finding Defects                    | 37 |
| Black Box & White Box Testing      | 37 |
| Performance, Load & Stress Testing | 38 |
| Quality Assurance                  | 38 |
| Implementation                     | 39 |
| Maintenance                        | 39 |
| Types Of Software Maintenance      | 39 |
| Types Of Software Support          | 41 |
| Roles Of SDLC                      | 41 |
| Business Roles:                    | 41 |
| Project Roles:                     | 42 |
| Technical Roles:                   | 42 |
| Implementation And Support Roles:  | 43 |
| Team Working & Structure           | 43 |
| Deliverables Of SDLC               | 44 |
| Diagram Deliverables               | 45 |
| Brief History Of UML               | 46 |
| Entity Relationship Models         | 46 |
| Class Relationship Models          | 51 |
| Use Case Diagrams                  | 55 |
| Use Case Descriptions              | 57 |
| Component Diagrams                 | 58 |
| State Transition Diagrams          | 60 |
| Sequence Diagrams                  | 62 |
| Activity Diagrams                  | 65 |
| Software Execution                 | 67 |
| High-Level Languages               | 67 |
| Translators                        | 68 |
| Assembly Language                  | 70 |
| Machine Language                   | 70 |
| Analogue VS Digital                | 70 |
| Introduction To C#                 | 71 |



|                                   |     |
|-----------------------------------|-----|
| What is C#?                       | 71  |
| Creating Variables                | 71  |
| Display Variables                 | 73  |
| Data Types & Conversion           | 74  |
| Take Input                        | 76  |
| Operators                         | 77  |
| Selection                         | 81  |
| The If Statement                  | 83  |
| The Switch Statement              | 85  |
| Iteration                         | 87  |
| Loops                             | 87  |
| Recursion                         | 91  |
| Methods                           | 96  |
| Collections                       | 100 |
| Arrays & Multi Dimensional Arrays | 100 |
| Stacks                            | 103 |
| Queues                            | 105 |
| Lists                             | 107 |
| Objects & Classes                 | 110 |
| Create A Class                    | 110 |
| Create A Object                   | 111 |
| Multiple Classes                  | 113 |
| Access Modifiers                  | 114 |
| Encapsulation                     | 114 |
| Properties Short-Hand             | 116 |
| Constructors                      | 117 |
| Inheritance                       | 120 |
| Polymorphism                      | 122 |
| Abstraction                       | 126 |
| Interfaces                        | 129 |
| Multiple Interfaces               | 131 |
| Namespaces                        | 132 |
| Design Patterns                   | 136 |
| Categories                        | 136 |
| Creational                        | 136 |
| Structural                        | 142 |
| Behavioural                       | 143 |
| Programming Principles            | 143 |
| Glossary                          | 144 |



# Foreword

---

This book was written for educational purposes, by Jade Lei ©Chairbear, for use by any individual or organization under the BY-NC-ND creative commons licence (Attribution + Noncommercial + NoDerivatives).

This book is particularly useful for those studying a range of qualifications, including software development and IT.

A breadth of topics are covered in this book, with more advanced topics only summarized; it is recommended to take in this basic coverage, and explore particularly heavy subject areas yourself.

Many topics covered include diagrams and some explanations related to these.

And strengths and weaknesses were applicable are summarized with short explorations ascertaining to such.

A glossary is provided, if needed, however for particular instances throughout the book footnotes instead will be utilized to give a more instantaneous understanding if needed.

A special thank you to my father, for always believing in me, and to Click for giving me the opportunity to learn software development with them.

We acknowledge that not everyone learns the same way, and so suggest you find additional resources that accommodate your learning style.

---

Jade Lei  
SEO Technical Specialist,  
Software Development Apprentice (2020)



# The Software Development Lifecycle

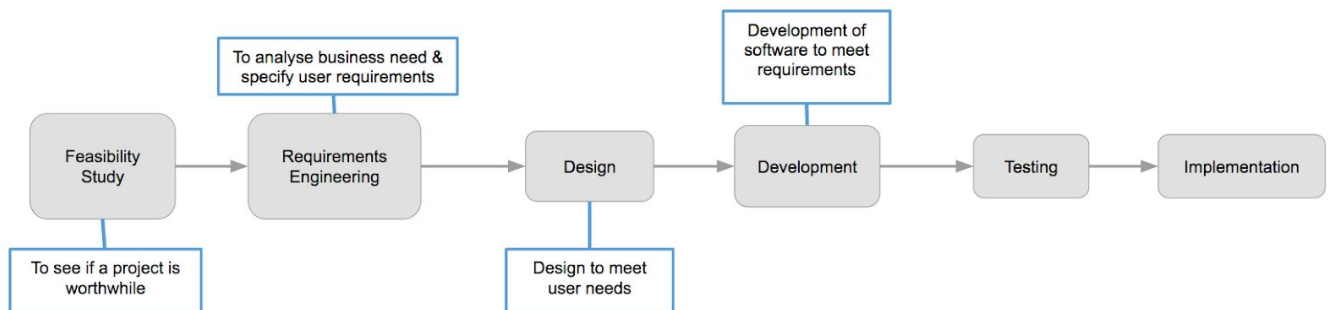
The Software Development Lifecycle (SDLC) is a framework describing a process for planning, building, testing and deploying a system. The process can apply to both hardware and software systems.

It aims to produce high-quality systems that meet or exceed customer expectations, reach completion within set times (deadlines) and within cost estimates.

## Why Develop New Systems?

In businesses, drivers for change are often regulatory change, business strategy change or an update in technology. For example, when the GDPR laws came into force, businesses had to update their websites to allow users to accept or decline cookies.

The financial standing of businesses relies heavily on their ability to compete with other companies in their industry, and so any aspect that can be changed in such a way that gives the company a competitive edge in the market has the potential to drive change. For example, updating their website to work on mobile devices, which will help attract new customers.



---

## Feasibility Study

Involves investigation and research in order to evaluate the projects potential for success to support decision making, and secure funding. It weighs the resources and costs involved against the project's value. This returns the return on investment (ROI).

### ROI Formula

$$\text{ROI} = (\text{Net Profit} / \text{Cost Of Investment}) \times 100$$

Only projects with a reasonable ROI will be supported.

It also identifies the resources available and those that need to be procured such as staffing, hardware and or software, and other assets such as premises.

There are different types of feasibility, explored further below:

### **Technical Feasibility**

- Will the proposed system perform to the required specification?
- Outline technical systems options proposed to use

### **Social Feasibility**

- Consideration of whether the proposed system would prove acceptable to the people who would be affected by its introduction
- Describe the effect on users from the introduction of the new system:
  - Will there be a need for retraining the workforce?
  - Will there be a need for relocation of some of the workforce?
  - Will some jobs become deskilled?
- Describe how you propose to ensure user cooperation before changes are introduced

### **Economic Feasibility**

- Consider the cost/benefits of the proposed system
- Detail the costs that will be incurred by the organisation adopting the new system: consider development costs and running costs

## **Requirements Engineering**

Aims to secure understanding in what the business needs the software to do. Requirements must be elicited, analysed, documented and validated by the business analyst.

The requirements are developed through 4 stages:

### **1) Elicitation**

- Elicitation practices typically include JAD (joint application development) techniques such as interviews, questionnaires, user observation, use cases, role-playing, and prototyping.
- This is likely to involve many different participants.

#### *Interviews*

##### *Strengths*

- Allows analysts to probe into greater depth
- Analyst can respond in real time to interviewee to ensure clarity of understanding

##### *Weaknesses*

- Time consuming and costly
- Subject to bias & interviews may produce conflicting results

#### *Questionnaires*

##### *Strengths*

- Allows collection of quantitative data
- Faster method of gathering from large sample size

#### Weaknesses

- Difficult to create well balanced questionnaire
- Difficult to receive good response rates

#### Observation

#### Strengths

- Identify peak and quit periods
- Check validity of information gathered in other methods

#### Weaknesses

- Participants often don't accurately recall everything they do
- People's behaviour changes when they are watched

### 2) Elaboration

- This stage provides greater depth to the reasoning of each requirement or user commentary, and breaks down each requirement or commentary into technical details.
- Methods used by the business analyst in this stage will include developing use cases, creating flow diagrams, class models, GUI mock-ups, and assigning business rules.
- This phase also assists with addressing known risk factors and establishes or validates the system architecture.

### 3) Validation

- This stage verifies that requirements are complete (and testable).
- The requirements document should be checked for ambiguities, conflicts and errors, while developers and testers should check the user commentary to ensure it matches up with criteria.
- Techniques used at this stage will include discussions, simulations, and face-to-face meetings.

### 4) Acceptance

- This is the final stage in requirements definition. It only happens when the requirements have been verified and agreed by all the stakeholders.
- It is during this stage that the business analysts create a 'baseline' of the requirements so that technical development can start and test planning can commence. This is done through an SRS (System/Software Requirement Specification) document which consists of all the product requirements to be designed and developed during the project life cycle.
- The functional and non-functional requirements will also be clearly specified here.

#### Requirement Levels

The acronym MoSCoW is often used to remember the levels of priority for requirements.

|          |             |   |
|----------|-------------|---|
| <b>M</b> | <b>MUST</b> | Describes a requirement that must be satisfied in the final |
|----------|-------------|---|



|          |               |   |
|----------|---------------|---|
|          |               | software solution   |
| <b>S</b> | <b>SHOULD</b> | Describes a requirement that should be included in the solution if possible, but can be satisfied in other ways if necessary                    |
| <b>C</b> | <b>COULD</b>  | Describes a requirement that is desirable but not a necessity. These are often included in the software if time and resources permit            |
| <b>W</b> | <b>WOULD</b>  | Describes a requirement that stakeholders have agreed will not be implemented in the current release but might be considered in future releases |

### Functional Requirements

In software development, a functional requirement defines a function of a system or its component. A function is described as a set of inputs, the behaviour, and outputs.

An example of a functional requirement would be a system that needs to be able to store user names in a database.

### Non-Functional Requirements

A non-functional requirement (sometimes referred to as NFR) is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviours. Such as speed or scalability.

Non-functional requirements are not any less important than functional.

An example of a non-functional requirement would be a system that should be able to carry out a thousand transactions per second. A non-functional requirement is almost like a SMART objective. It is measurable.

Many requirement gathering techniques focus on gathering functional requirements, however one way to ensure that as few as possible NFR are left out is to use NFR groups. The most common groups are : usability, reliability, performance and supportability.

### *Usability*

Important functions of the system are prioritized based on usage patterns. A non-functional requirement should be created to be able to test frequently used functions for usability. As well as non-functional requirements being created to test the usability of complex or critical functions.

### *Reliability*

Users rely on the system and must be able to trust it. The goal of an end system should be a long mean time between failures (MTBF). A requirement should exist that specifies that data created in a system should be retained for a number of years without the data being changed. Another recommended reliability requirement is one that can monitor system performance.

### *Performance*

These will be requirements that answer the following questions.

- What should the system response times be. As measured from any point?
- Under what conditions?

Performance requirements will be mindful of peak performance times (when the load of the system will be high), and stress periods.

### *Supportability*

A system must be cost efficient to maintain, and so maintainability requirements may cover levels of documentation such as system or test documentation.

Below is a non extensive list of typical non functional requirements.

|                    |                     |                                    |                        |                          |                    |                        |
|--------------------|---------------------|------------------------------------|------------------------|--------------------------|--------------------|------------------------|
| <i>Performance</i> | <i>Availability</i> | <i>Reliability</i>                 | <i>Maintainability</i> | <i>Disaster Recovery</i> | <i>Efficiency</i>  | <i>Fault Tolerance</i> |
| <i>Security</i>    | <i>Capacity</i>     | <i>Data Integrity</i>              | <i>Usability</i>       | <i>Interoperability</i>  | <i>Privacy</i>     | <i>Portability</i>     |
| <i>Quality</i>     | <i>Resilience</i>   | <i>Reliability &amp; Stability</i> | <i>Response Times</i>  | <i>Robustness</i>        | <i>Scalability</i> | <i>Testability</i>     |

### Further Example

*A page has a video on it. There should be a facility for the user to change the video audio volume. The audio should be controlled by a slider.*

The **functional requirement** will be the ability to change the audio.

The **non-functional requirement** will be the slider. This is to do with the GUI, it is a matter of choice, there are multiple options available to change the volume, you could have just as easily chosen a knob.

If a system calls a function to make something happen, that is a functional requirement, how you make the function happen is not functional, as it is a matter of choice.

Ei; the slider being moved to increase the audio is non functional - it could have just as easily been a knob turned up - however, the audio increasing as a result is functional.

#### *Functional VS Non-Functional Summary*

Functional requirements describe what a system should *do*, and specify the system's functions (features) whilst non-functional describe how it should *work*, and specifies the systems quality characteristics.

#### Additional Examples

| Functional   | Non-Functional   |
|--|--|
| Display records from a database  | Shown database data should be refreshed every 10 seconds |
| Clicking a send button will add the given user data to a user database | You can add a maxim of 10 users                          |

## Design

The software requirements specification (SRS) from the requirements stage is the reference for product architects to decide the best architecture for the product to be developed. Based on the requirements specified in the SRS, usually more than one design approach is proposed and documented in a Design Document Specification (DDS).

This DDS is reviewed by stakeholders and a approach is selected based on variables such as:

- risk assessment
- product robustness
- design modularity
- budget and time constraints

The chosen design is elaborated upon. Prototypes may be produced as a proof of concept.

#### The Purpose Of Design

Software design is the process of understanding a problem, and planning out a way to solve it using software. Software design should:

- Aid communication between 'actors'
- Be used as a basis for rigorous development
- Provide a standard approach
- Ensure consistency across the development
- Assist in the identification of re-use
- Be used to compare the current situation with the required

There are 2 main types of design: Logical and Physical.

### *Logical*

A logical design states what must be done, but not how it will be done. A logical design doesn't address the actual methods of implementation.

An example of logical design is written descriptions, or modelling notations such as use cases.

### *Physical*

Physical design is concerned with the implementation, it is the how, how the software is going to achieve it's requirements. Data, and process design and UI design are part of physical design.

### *Algorithms*

A program is a set of instructions that describe how to solve a problem, written in a language that a computer can understand. However, complex solutions cannot be written straight away, they must be designed and the logic of these designs checked to ensure that the software will meet the requirements and solve the problem. This is where algorithms come into play.

An algorithm is a simple set of rules or instructions that a computer follows to complete a task. It is the steps involved to provide a solution to a particular problem. Algorithms can be used in both logical and physical design.

### Characteristic Of Algorithms

- Precision
- Finiteness
- Speed

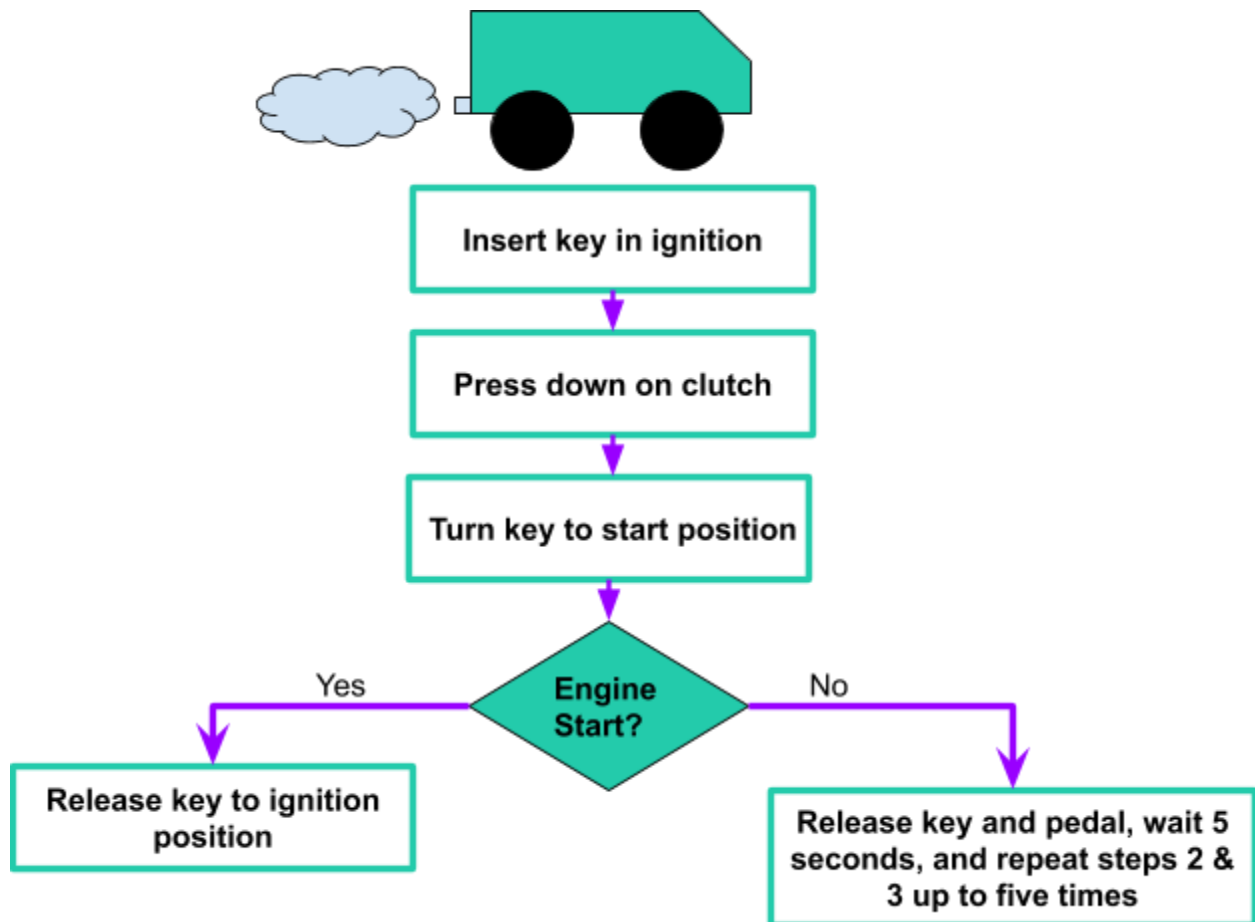
The Big O notation is the official notation to describe the performance of algorithms. It describes the performance or complexity of an algorithm under the worst-case scenario.

### General Rules Of Algorithms



- Inputs & outputs should be defined precisely - steps should be clear and unambiguous (Precision)
- The algorithm should solve the problem and terminate (Finiteness)
- The algorithm should aim to solve the problem in the quickest & most efficient way (Speed)
- The algorithm should be written in such a way that it can be used in any programming language (Language Independent)

### *Example Of Algorithm*



Algorithms written in english can be ambiguous, as text can be subjective. This can be disastrous for program design as we potentially may end up with the wrong software being created. Therefore, there are more valid ways of writing algorithms:

- Pseudo Code
- Flowcharts
- [UML Activity Diagrams](#)

Written form is only used if the program is simple and the language used is clear. Other valid forms of representing algorithms is any modeling notation that allows explicit processing and logic to be shown for maximum clarity.

### *Pseudo Code*

Pseudocode is a high level descriptive explanation of code. It is neither normal english or actual code. It is sometimes referred to as structured english. The idea of it is that it can be used by any programmer to create a bit of code in any language.

| Pseudo Code   | Actual Code (C# Example)  |
|---|---|
| Input student grade<br>If grade >= 60<br>Output "passed"<br>Else<br>Output "failed" | <pre>int grade =<br/>Convert.ToInt32(Console.ReadLine());<br/>if (grade &gt;= 60){<br/>    Console.WriteLine("passed");<br/>} else {<br/>    Console.WriteLine("failed");<br/>}</pre> |

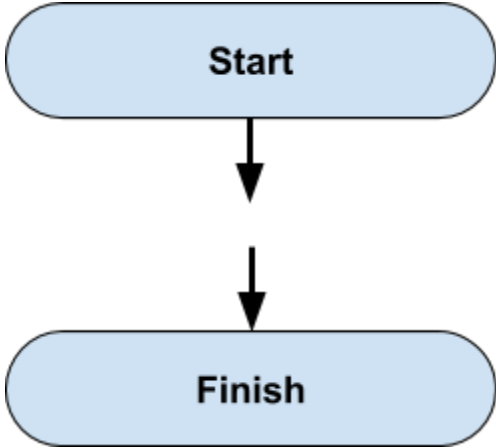
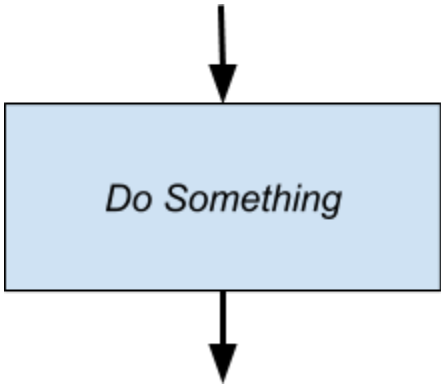
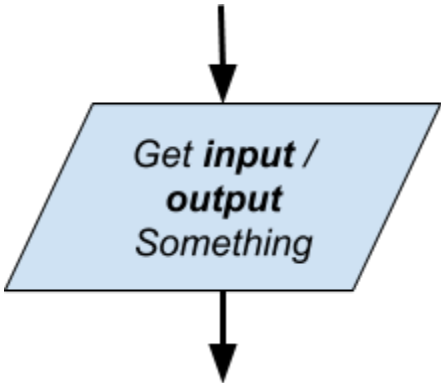
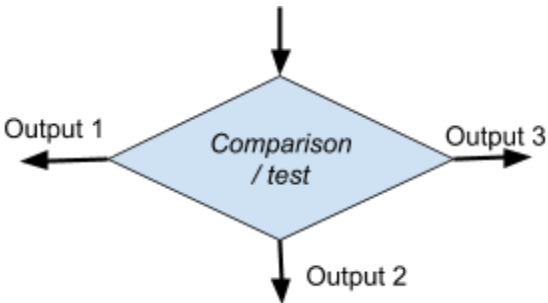
Although there are no official rules or notation for pseudocode, it should be written so that it is coding language independent, and statements showing dependencies are indented. Such as if, while, for and switch.

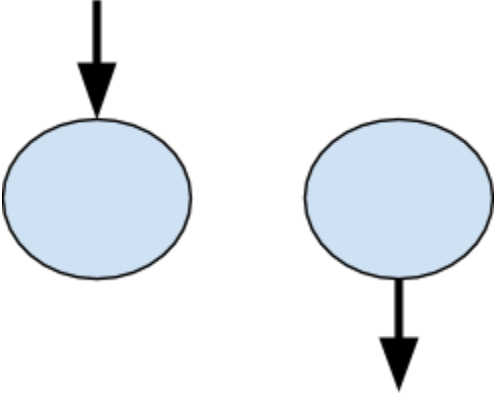
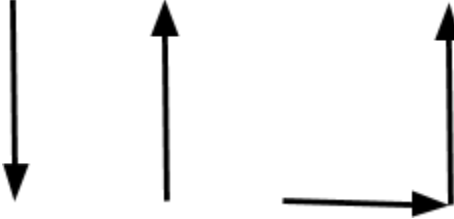
### *Flowcharts*

A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem.

### Flowchart Notation & Guidelines

| Notation | Meaning |
|----------|---------|
|----------|---------|

|   |  |
|---|--|
|  <pre> graph TD     Start([Start]) --&gt; Arrow1[ ]     Arrow1 --&gt; Finish([Finish])   </pre>  | <p><b>Initiators &amp; Terminators</b></p> <p>Show the start or end of a program</p> <p>The usual direction of the flow is from left to right, or top to bottom</p>  |
|  <pre> graph TD     Arrow1[ ] --&gt; Process[Do Something]     Process --&gt; Arrow2[ ]   </pre>  | <p><b>Process/Function</b></p> <p>Shows computational steps, or processing function of a program</p> <p>Only one flow line should come out of the process symbol</p> |
|  <pre> graph TD     Arrow1[ ] --&gt; IO[/Get input /<br/>output<br/>Something/]     IO --&gt; Arrow2[ ]   </pre>   | <p><b>Input / Output Operation</b></p>   |
|  <pre> graph TD     Arrow1[ ] --&gt; Decision{Comparison / test}     Decision --&gt; Out1[Output 1]     Decision --&gt; Out2[Output 2]     Decision --&gt; Out3[Output 3]   </pre> | <p><b>Decision Making Branch</b></p> <p>Only one line should enter the decision branch but 2-3 should leave - one for each answer</p>                                |

|   |   |
|---|---|
|  | <p><b>Connector</b></p> <p>Used to join two parts of a program together</p> |
|  | <p><b>Flow Lines</b></p>  |

### Advantages Of Flowcharts

- A clear representation of steps and outcomes, for clarity of communication
- Suitable for analysis and documentation
- Efficient for coding
- Allows for debugging and maintenance

### Limitations Of Flowcharts

- Not good for complex logic, as the chart will get crowded and become hard to read
- If alteration is required the chart may have to be redrawn completely

### *Types Of Algorithm*

In programming, there are many common algorithms and techniques used to solve commonly encountered problems. Of these, we will look at sorting, searching and encryption algorithms.

### Sorting Algorithms

#### **Bubble Sort**

In the bubble sort, adjacent pairs of numbers are compared, and if they are the wrong way around (in numerical order) then they are swapped. This moves the highest number 'bubble' to the end of the list. When the end of the list is reached, the process begins again at the start of the list and the process is repeated until there are no swaps made, and the list is then considered sorted.



## Quick Sort

This algorithm uses a divide and conquer method in a recursive manner, and chooses a pivot point in a collection of elements.

It partitions the collection of elements around said pivot point, so that elements smaller than it are moved before it, and the larger elements are moved after it.

## Insertion Sort

At each iteration, insertion sort removes one element from the unsorted part of the list, finds the location it belongs to in the sorted list, and inserts it there. This repeats until no input elements remain.

## Selection Sort

Selection sort scans the unsorted list for the next element to include in the sorted subarray. It will swap the next item in the list for the minimum value in the unsorted list.

## Searching Algorithms

A searching algorithm is a method of locating a specific item in a larger collection of data.

## Sequential Search

The sequential search algorithm is a technique for searching the contents of an array. It uses a loop to sequentially step through an array, beginning with the first element, and compares each element with the value being searched for.

The search ends when the value has been found or when the search algorithm reaches the end of the array (no value has been found).

## Binary Search

For binary search to work, the array has to be sorted in ascending order. The data is then examined, and the half way point found. If the value we're looking for is not below or at the halfway point, that half of the data is discarded. The process is repeated until the value is found, or all data is discarded.

Binary search is one of the most efficient algorithms, because in each iteration we half the search space.

## Encryption Algorithms

Encryption is based on the science Cryptography which consists of making things cryptic. Encryption allows us to encrypt data into seemingly unreadable gibberish, and decrypt it to be able to read it. A key (also known as a cipher) is required to decrypt the data. There are two major types of encryption: symmetric and asymmetric.



The major difference between the two is that symmetric encryption uses one key for both the encryption and decryption whilst asymmetric encryption uses a public key for encryption and a private key for decryption.

### Symmetric

- The preferred technique for transmitting data in bulk due to the algorithm being less complex than others and having a fast execution time
- Plaintext is encrypted using a key, and the same key is used on the receiving end to decrypt the received ciphertext

### Asymmetric

- More secure than symmetric, but takes longer
- The public key used for encryption is available to everyone but the private key is not disclosed
- When a message is encrypted using a public key it can only be decrypted using a private key, however, if a message is encrypted using a private key it can be decrypted using a public key

Some examples of common encryption algorithms are:

| Name                    | Description   |
|-------------------------|---|
| DES / 3DES or TripleDES | Data Encryption Standard is commonly used in ATMs and UNIX password encryption. 3DES & TripleDES replaced older versions as a more secure method of encryption, as it encrypts data three times and uses a different key for at least one of the versions |
| Blowfish                | A symmetric block cipher  |
| AES                     | Advanced Encryption Standard uses the Rijndael block cipher. Replaced by DES in 2000  |
| Twofish                 | Is a block cipher   |
| IDEA                    | Features 64 bit blocks with a 128 bit key   |

Fun fact: an early form of encryption was used by Julius Caesar. Caesar shifted each letter of the alphabet to the right or left by a number of positions. This technique is called Caesar's cipher.

### Categories Of Algorithm

- Simple Recursive Algorithms



Many algorithms use the concept of recursion - which is to call a part or all of themselves repeatedly. A simple recursive algorithm solves the base cases directly, recurs with a similar subproblem and does some extra work to convert the solution to the subproblem into a solution to the given problem.

- Backtracking Algorithms

Are based on a depth first recursive search. They test to see if a solution has been found, and if so, returns it otherwise for each choice that can be made at this point, a choice is made, recursed, and if the recursion returns a solution, it is returned. If no choices remain, a failure is returned.

- Divide And Conquer Algorithms

A divide and conquer algorithm consist of two parts:

1. Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively
2. Combine the solutions to the subproblems into a solution to the original problem

Mergesort and quicksort are examples of a divide and conquer algorithm.

- Dynamic Programming Algorithms

A dynamic programming algorithm remembers past results and uses them to find new results. It is generally used for optimization problems, where multiple solutions exist, but you need to find the "best" one.

DP requires "optimal substructure" and "overlapping subproblems".

Optimal substructure is where the optimal solution contains optimal solutions to subproblems. And overlapping subproblems is where solutions to subproblems can be stored and reused in a bottom-up fashion.

This differs from divide and conquer, where subproblems generally need not overlap.

- Greedy Algorithms

A greedy algorithm works in phases, and at each phase:

- You take the best you can get right now, without regard for future consequences
- You hope that by choosing a local optimum at each step, you will end up at a global optimum

- Brute Force Algorithms

A brute force algorithm simply tries all possibilities until a satisfactory solution is found. There are two sub-types:



- OPTIMIZING: Find the best solution. This may require finding all solutions, or if a value for the best solution is known, the algorithm may stop.
- SATISFYING: Stop as soon as a solution is found that is good enough.
- Randomized Algorithms

A randomized algorithm uses a random number at least once during the computation to make a decision. Such as in quicksort, using a random number to choose a pivot.

### *Computational Thinking*

Computational thinking allows us to use computers to help solve complex problems. It allows us to understand what the problem is, and develop possible solutions, which we can then present in a way that a computer and human can understand.

Computational thinking is composed of four key techniques, one of which we have just discussed:

- Algorithms

Developing a step by step solution to the problem, or the rules to follow to solve the problem.

- Decomposition

Breaking down a complex problem or system into smaller, more manageable parts

- Pattern Recognition

Looking for similarities among and within problems

- Abstraction

Focusing on the important information only, and ignoring the irrelevant detail

### *Decomposition (Functional Decomposition)*

Involves breaking a process down into smaller parts/functions, and then breaking them down further into smaller simpler parts.

### *Pattern Recognition*

Also known as pattern matching. It is a form of computational thinking that identifies whether two or more processes , problems, or data are similar. Pattern recognition involves finding similarity among small, decomposed problems that can help us solve more complex problems, more efficiently.

A commonly used programming technique for pattern recognition is regular expression (RegEx).



A regular expression is a sequence of characters that define a search pattern.

### *Abstraction*

Abstraction is a term that can have different meanings in different contexts, however in this context it refers to removing unnecessary details.

### *Design Techniques & Documentation*

As well as the DDS - also referred to as the technical design specification or design specification - these documents may contain a variety of different design works and contents including:

- Data Design  
    Ei ERD, Data Dictionaries
- UI Design  
    Ei wireframes, moodboards
- Architecture Design  
    Logical models of different software layers and the networks/hardware involved
- Process Design  
    The proposed logic and flow of code using flowcharts, activity diagrams, pseudocode

### *Software Architecture*

Software itself has many different architectures and designs, with modern software developments using a multi-tiered approach where the software code itself is split into different components – each with its own purpose and functionality.

Benefits of tiered software architecture are:

- Increased Maintainability & Supportability
- Faster Code Development & Modular, Reusable Code
- Multiple Developers Can Work On Different Components In Parallel

The 3 tier architecture layers are:

- 1) User Interface  
    This layer is the code for the interface/GUI. It is sometimes referred to as the presentation layer
- 2) Business Logic



This layer is where the processing of UI actions and data is performed according to the business rules

3) Data

This layer is where the data is stored (e.g. in a database), and code interacts with the stored data

## Inputs To Solution Architecture

As solution architecture is focussed on the implementation of business objectives, inputs at this level are related to the needs of the business and its stakeholders.

*Business Drivers And Need*

*User Requirements*

*Budget And Resources*

## Infrastructure Architecture

Also known as technology infrastructure, this architecture is concerned with modelling the hardware elements within an organisation, and the interaction and relationships between them. This includes client devices, servers, all networking and communications equipment, and the location of databases. Without infrastructure architecture, the other higher-level architectures would not be possible, as it is the basis for organisation and system communications, the hosting point for applications, and the connection to external stakeholders and systems.

## Inputs To Infrastructure Architecture

These include:

*Solution/Software Capacity & Performance Requirements*

*User Needs*

*Budgets And Resources*

*Technologies Available*

## Data Modelling

Data modeling is the exploration of data oriented structures, and like other modelling techniques explored further in Deliverables, data models can be used for purposes from high level conceptualization to physical data models. Physical data modeling is conceptually like class modeling, with the goal being to design databases - depicting the tables, the data column in those tables, and the relationship between said tables.

There are 3 types of data models:

- Conceptual

This model defines what the system contains, its purpose is to organise, scope and define business concepts and rules. It is typically created by the business stakeholders and architects.

- Logical

This model defines how the system should be implemented regardless of the database management system(DBMS), its purpose is to develop a technical map of rules and data structures. It is usually created by the architects and business analysts.

- Physical

This model describes how the system will be implemented using a specific DBMS system, the purpose being the actual implementation of the database. This is created by developers and the database administrators.

## Interface Design

User interface (UI) design is critical to the success of software, as it is the main point of interaction from the user and one of the main driving forces behind users' subjective opinions on a software's quality.

Interface design normally refers to designing the layout, styling, text, controls, flow and usability of visual screens.

When designing the software there are two main ways to model the interface design, these are wireframing and [prototyping](#).

A wireframe is a visual layout of how a user interface (UI) should look. They include basic information such as the position of elements like images or forms, and can be anything from a rough sketch to a detailed plan.

Other methods include mockups and moodboards.

Mock-ups are similar to wireframes in that they are a constructed image that shows what a final UI may look like, however they are usually produced using graphical software and include colors, fonts and images that a wireframe does not.

Mock-ups are sometimes chosen over wireframes because:

- Able to demonstrate a 'mock-up' of design to client before an ounce of coding is done
- They provide a better understanding of how the UI will look
- Assets and styles that will be used in live project are created and showcased

Mood boards on the other hand are a collection of assets and materials intended to communicate the style, voice, or direction of a particular brand or project.

Storey boarding is another method, however is not particularly popular for software design, more video creation. It consists of visual representation of a sequence (action) and breaks it down into individual panels. The sketches will show how the sequence unfolds, shot by shot.



For example, a user highlights some text on a site, when the text is highlighted share icons pop up, the user can then select a share tool and select the location to upload.

### *Design Principles*

Good design should follow these principles:

*No Wasted Space, But No Cluttering Either*

*Don't Give An Unnecessarily Large Number Of Options (Hick's Law<sup>1</sup>)*

*Use Of Appropriate Colors (Psychology Of Colours<sup>2</sup>)*

*Make Controls Large, And Easy To Reach (Fitts' Law<sup>3</sup>)*

### *Data Design*

Often, software is required to persist it's data, this means store it on a permanent basis so that it can be retrieved and processed again later. Software can do this using files or databases.

This is part of the physical design process. For files, you need to consider the structure or format of the data such as JSON, CSV, ect. For databases, it is likely that normalisation<sup>4</sup> and creation of a data dictionary or entity relationship diagram will be needed.

### *Data Dictionary*

Is a set of information describing the contents, format, and structure of a database and the relationship between its elements. It is used to control access to, and manipulation of, the database.

### *Entity Relationship Diagram*

An ERD consists of entities, attributes and relationships, and it is a graphical representation of entities and their relationships with one another. See more [here](#).

### *Design Constraints*

There are obvious constraints such as performance capabilities of the hardware, time constraints and staffing limitations. However there are 3 main restrictions that mold a system design, these are:

---

<sup>1</sup>

<https://www.interaction-design.org/literature/article/hick-s-law-making-the-choice-easier-for-users>

<sup>2</sup> [https://en.wikipedia.org/wiki/Color\\_psychology](https://en.wikipedia.org/wiki/Color_psychology)

<sup>3</sup> <https://www.interaction-design.org/literature/topics/fitts-law>

<sup>4</sup> Is the process of organising columns (attributes) and tables of a database to ensure that their dependencies are properly enforced by database integrity constraints.

[https://en.wikipedia.org/wiki/Database\\_normalization](https://en.wikipedia.org/wiki/Database_normalization)





## Legal

These are constraints on development enforced by law. One example would be 2018's Data Protection Act (GDPR).

## Ethical

Ethical refers to moral values and considering what is generally thought to be good practise. One such 'good practise' is developing with the public interest in mind.

## Financial

This means there will be a budget to stick to. Things that fall under finances include staff and resource cost. Often cost/benefit analysis is performed, and expenditure monitored at particular development phases (depending on methodology used).

### An Example Of Constraints In Practise

A business is developing some email marketing software that allows users to make a marketing email using a template and send it to their emailing list. Funds are running a little low, and so the product is packaged and sold without fully being complete - an email unsubscribe button is left missing out of the templates.

The **ethical issue** here is that the company knowingly is selling an uncompleted product but still selling it as having all the bells and whistles. The **legal issue** is that email recipients have no way of unsubscribing from the emails.

### *Building In Security*

Design also incorporates consideration of quality, security and constraints. Often referred to as 'building security in.' This is the process of adding security to the development of software at an early time.

This consideration leads to secure development. Which is a practice to ensure that the code and processes that go into developing applications are as secure as possible. It includes:

- Threat modeling

This asks what parts of the software are easily compromised, and what is the impact?

Threats will be prioritised and if a threat cannot be mitigated with security control the design is changed or the risk is assumed.

- Testing

This can be static code analysis in which code is analysed without running, or testing the software against a modelled threat.

There is even a SDLC dedicated to secure development called the Security Development LifeCycle that states security activities must be integrated into every phase of development. It originates from Microsoft.

| Training               | Requirements   | Design  | Implementation   | Verification  | Release  | Response                       |
|------------------------|--|---|--|---|--|--------------------------------|
| Core Security Training | Establish Security Requirements<br><br>Create Quality Gates / Bug Barriers<br><br>Security And Privacy Risk Assessment | Establish Design Requirements<br><br>Analyse Attack Surface<br><br>Threat Modelling | Use Approved Tests<br><br>Depreciate Unsafe Functions<br><br>Static Analysis | Dynamic Analysis<br><br>Fuzz Testing<br><br>Attack Surface Review | Incident Response Plan<br><br>Final Security Review<br><br>Release Archive | Execute Incident Response Plan |

*Pro-active Security Approaches Summary Table*

|   |  |
|---|--|
| <b>Defensive Programming</b>                      | Programming whilst incorporating exception handling, validation and security features such as authentication. The worst-case scenario is imagined and code is created to handle or avoid it. |
| <b>Permission Setting &amp; Role Based Access</b> | Creating permissions for access to files and data, or setting different user access levels. This is authorization.   |
| <b>Authentication</b>                             | Require all users of systems to be authenticated before granting access.   |
| <b>Encryption</b>                                 | Use strong encryption algorithms to 'scramble' data and transmissions between devices.   |

## Threat VS Vulnerability

A threat is something that is likely to cause damage or danger - in this instance, to a system- whilst a vulnerability is the state of being exposed to a possible threat. Risks are the result of having a vulnerability which could be exploited by a threat. Impact is the marked effect a threat has.

Threats in IT can be categories into two types:

- Accidental
- Deliberate

With these also having sub-categories of internal and external threats.

## **Security VS Resilience**

Security is the pro-active protection against threats, for example firewalls and antivirus software, whilst resilience (also known as threat mitigation) is the reactive countermeasures that limit the damage an attack has on a system and infrastructure, for example system backups.

## **Common Security Attacks**

### **Brute Force Attack**

Often referred to as existing in the 'hacking' genre of security attacks, this method consists of offline attacks with passwords. It assumes most passwords are found in cracking dictionaries, and the attacker relies on high volume guessing. It is a form of password API.

### **Credential Stuffing**

Is another in the 'hacking' genre, however this method is an online attack with credentials. It assumes most people reuse the same credentials on multiple sites, and the attacker relies on breached credentials. It is a credential API.

### **Malware**

Is malicious software. It is any software intentionally designed to cause damage to a computer, server, client, or network. Examples include trojan horse, and spyware.

### **Cross Site Scripting (XSS)**

Typically found in web applications, XSS enables attackers to inject malicious client side scripts into web pages viewed by other users. A XSS vulnerability may be used by attackers to bypass access controls such as the same origin policy.

#### Example

- 1) An attacker discovers a XSS vulnerability on a website

This is usually a user input field that has not been properly configured and restricted which enables attackers to place scripting markup within the input area and run it.

Preventing this is dependent on the language behind the website.

For example, with PHP you can use PHP's htmlspecialchars() function to prevent this.



Whilst using htmlspecialchars() if a user tried to submit the following in a text field:

```
<script>something</script>
```

It would not be executed as it would be saved in HTML escaped code like this:

```
&lt;script&gt;something&lt;/script&gt;
```

- 2) The attacker then injects a malicious script that steals each visitor's session cookies - which will say something to the computer akin to 'this is a valid user'
- 3) The attacker can then use these cookies to gain access to whatever privileges the original user has. If they were admin, for example, the attacker would then have access to any important or sensitive information the administrator had access to

## Cross Site Request Forgery

Unlike XSS which exploits the trust a user has for a site, CSRF exploits the trust a site has in a user's browser. CSRF is a type of malicious exploitation of a site where unauthorised commands are transmitted from a user that the application trusts, without the user's interaction or knowledge.

### Example

- 1) Jem receives an email from a stranger that asks if they have seen the most recent news about a breaking news robbery in their country. The email contains a link that appears to go to a news site
- 2) Jem clicks the link
- 3) The link itself now attempts to access funds in Jem's bank account, and since the request comes from Jem's computer and it has cookies authenticating them the link has access and successfully transfers funds to another bank account

## SQL Injection

An infamous type of attack, SQL injection is a code injection technique similar to Cross Site Scripting in which nefarious SQL statements are inserted into entry fields to attack data driven applications. For example, to send all the database contents to the attacker.

### Example

A basic login box:

|           |                          |
|-----------|--------------------------|
| Username: | <input type="text"/>     |
| Password: | <input type="password"/> |

The SQL Injection:

|           |   |
|-----------|---|
| Username: | <input type="text" value="' OR 1=1; /*"/> |
|-----------|---|



Password:

What this means to the database:

```
SELECT * FROM Users WHERE Username='' OR 1=1;/* AND Password='*/--'
```

Translation:

Select all from the Users table where Username is equal to nothing or equal to true (to the database, if a true value is passed it means 'a value was found'), and Password is cancelled out as the \*/-- turns the rest of the statement into a comment.

An easy fix is to prevent the input boxes from accepting special characters.

### **Denial Of Service Attack (DoS)**

In a DoS attack, the attacker aims to make a machine or network resource unavailable to its intended users by either temporarily or permanently disrupting the services of a host. It is typically accomplished by flooding the targeted machine or source with a staggering amount of requests in an attempt to overload the system, which will prevent legitimate requests from being fulfilled.

There are also attacks that focus on manipulating a person into divulging information or access to a computer system, this is social engineering.

Below are a few examples of this.

### **Phishing Emails**

This involves getting users to divulge information by tricking them with fictitious emails.

*Whale phishing* includes targeting a more profitable user (someone who is important or wealthier than a standard target). A juicer target if you will. For example, a CEO of a company instead of a receptionist.

*Spear phishing* includes using user specific emails, to make users feel it is more reliable. Such as using your (the victims) name.

### **Pretexting**

This includes using a plausible scenario to gain people's trust.

### **Water Hole Attack**

This includes targeting social websites or online gathering places for a particular sort of person.

As the attacker knows the general subject or topic being discussed here, they can find it easier to coin their attacks by playing to the users weaknesses.



For example, a football fan forum, and the attacker posting a message saying they need money to support their son's football team. They could follow the message up with a plea for other football fans to send money to support the non existing team.

## Development

Is the phase where technical components<sup>5</sup> are created, procured or configured.

To build in security developers must work with testers and security experts to better understand their code from an attackers point of view. They should also train in secure coding practices and keep up-to-date with the latest cyber attacks and vulnerabilities. Any software that requires user input should also be validated.

### *Proactive Security Approaches*

4 examples of proactive approaches are:

#### Defensive Programming

Programming that incorporates exception handling, validation and other security features. The worst case scenario is imagined, and then coding measures put in place to try to prevent it.

#### Permission Settings

This incorporates creating permissions for access to files and data, and setting different user access levels.

### *Authentication VS Authorization*

Authentication is when a user proves who they say they are. Whilst authorisation determines what that user is and isn't able to do.

User activities are often tracked, to be able to enforce their accountability for their actions. This is called logging or recording.

### *Authentication Factors*

---

<sup>5</sup> A part or element of a larger whole. *Analogy - a single puzzle piece that connects with others to make a whole picture.*



There are three factors for authentication, each with their own strengths and weaknesses.

- Know

Something a user should know, such as a password. Also known as authentication by knowledge, it is the least expensive to implement, however the downside is that another person may gain access to this information, and therefore unauthorized access.

- Have

Something a user should have, such as a card. Mid level expensive. Depending on the item the user should have, this provides a stronger level of protection as someone can easily identify that one user's face on an ID card is different to that of another person. However, again depending on the item, this method can also be insecure, as a user could lose a physical item and another unauthorized person gain access to it.

- Are

Something users are, such as fingerprints. The most expensive, but most secure level of authentication.

Biometrics verify an individual's identity by analysing a unique personal attribute or behaviour. This method can be difficult as it relies on humans being the keys themselves, and this leads to issues such as with aging or biological defects. As some people do not have fingerprints, so fingerprint scanners will not work for them. A retina scanner relies on a user's eyes, but cataracts and other eye conditions can cause the system to misidentify users. Hand scanners that rely on light reflection also rely on the user placing their hand on the scanner in the same way every time for authentication, which is almost humanly impossible as human stance, etc. rely on many factors itself, and so these machines often work with a high level of % variation.

### *Authorisation Matrix*

An authorisation matrix is a list of roles, users, and permissions that these roles/users have. It sets out who can make what changes. Such as setting up a new user and giving them read one access to a database.

### Physical Infrastructure

This includes utilizing physical measures of protection such as biometrics, CCTV and locks. Physical security often can be split into two categories:

#### *Deterrence*

Such as warning signs, fences, and guards.

#### *Delaying & Detection*

Such as locks, cameras and alarms.

## Policies

It is common practise that organisations have policies and procedures for security. Policies set out the long term, high level objectives in regards to the organisations IT security, whilst a security procedure is a set of necessary activities that perform a specific security task or function.

## *Firewalls*

Firewalls are set up with firewall policies which set up the rules as to what traffic can or cannot pass through.

Firewalls can have defaults:

- Default permit - everything allowed through unless expressly forbidden.
- Default deny - nothing allowed through unless expressly permitted. The firewall rules will revolve around opening or closing PORTS.

PORTS are a software connection point or interface. When data packets are sent through networks (usually via TCP/IP), the source and destination IP addresses are on each packet. The packet header will also include the PORT. This is so that the receiving machine knows where to send the data when it is received. A firewall can examine each data packet and block or allow packets through depending upon its destination address, port number or even packet contents.

Common PORT numbers include:

- 80 – HTTP. This is web traffic
- 25 – SMTP. This is email.
- 110 – POP3. This is also email.
- 443 – HTTPS. This is secure web traffic.
- 20/21 – FTP. This is file transfer.

All of the above use the TCP protocol.

## TCP VS IP

TCP stands for transmission control protocol, and it is a standard that defines how to establish and maintain a network conversation through which programs can exchange data. IP stands for internet protocol, and it works with TCP. IP defines how computers send packets of data to each other.

## Types Of Firewall

There are many different types of firewall, but the most common include:

- Packet Filtering firewalls
- Stateful Inspection firewalls





- Circuit gateway firewalls
- Proxy (Application) Firewall

### *VPN*

VPN stands for virtual private network. A VPN creates an encrypted 'tunnel' between the two ends of the connection. It essentially extends a private network across a public network to enable users to send and receive data across shared or public networks as if their computer was directly connected to the private network.

You can set up a VPN connection on your router, however many 'third-party' companies offer to set up a VPN for you to use. Although they do a lot of the technical work for the user, the difference between your own connection on your router and a third party is that all traffic goes through the servers of the third party.

### *DMZ*

Stands for demilitarised zone. A DMZ protects a contained sub-network from the threats of a larger, untrusted network, such as the Internet.

It uses your router to move part of your private LAN<sup>6</sup> so it is more publically accessible. This allows the outside world easier access to web servers, games consoles etc.

Although this does reduce the security of the devices in the DMZ, it is often used to reduce problems caused by the firewall being too restrictive.

DMZ is considered better than PORT forwarding, as all traffic on all ports for the demilitarised devices will be sent straight there, vs redirecting a communication request from one address and port number combination to another while the packets are traversing the network gateway.

### *Prototyping*

Is the activity of creating early sample versions of software. Typically simulating only a few aspects of the final product. There are two models for prototyping, throwaway model and evolutionary model.

#### *Strengths*

- Confusing, missing or difficult functions can be identified
- Errors can be detected early

#### *Weaknesses*

---

<sup>6</sup> Stands for local area network. It is a computer network that interconnects computers within a limited area such as that of a company.



- Leads to 'implement & then repair' way of working
- May increase complexity of the system as scope may expand beyond original

- Throwaway Model

This prototype is thrown away once it has been reviewed. They are quick to produce, and often visually resemble the final product but have no to little functionality

- Evolutionary Model

This prototype evolves into the final product. The main goal is to build a very robust prototype in a structured manner and then refine it. The prototype forms the heart of the new system and the improvements and further requirements will then be built

## Testing

Components are tested to ensure they meet requirements. Levels of testing include unit, component, integration, system and acceptance and regression testing.

### *Unit Testing*

Is a development process in which the smallest testable parts of an application, called units, are individually and independently scrutinised for proper operation.

Unit tests are usually written and performed by developers to ensure code meets its design and behaves as expected. Unit testing should be done before integration testing<sup>7</sup>, and can be done via black or white box testing methods.

### *Component Testing*

Similar to Unit testing, component testing is performed on each individual component separately without integrating with other components.

### *System Testing*

Is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

Often carried out by independent or specialist testers.

### *Acceptance Testing*

The purpose of this testing level is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

---

<sup>7</sup> Testing of combined components to determine if they function correctly.

Acceptance testing is done by the user/customer. And it can be separated into Alpha<sup>8</sup> and Beta<sup>9</sup> testing.

### *Regression Testing*

This testing is done to ensure that every subsequent release or iteration of development does not break anything that was previously working.

## **The General Testing Process**

1. Planning

This establishes the the scope of testing, and acceptance/exit criteria

2. Analysis & Design

This is the designing of the test cases

3. Implementation & Execution

4. Evaluation

Assessing if the testing is complete and reporting of the results

## **Exit Criteria**

Exit criteria is used to determine whether a given test activity has been completed or not. For example, verifying that all tests planned have been run, if so then this exit criteria has been met. When criteria is met, a test summary report will be generated and shared with stakeholders. From this the decision may be made to go onto a new test level or finalise this as the last testing stage.

## **Why Test Early In Development**

Problems that are introduced into the system during design or planning can be caught, and the requirements testing can anticipate future problems at a lower cost.

Quality will also be ensured. Test cases written during requirements gathering can be shared with developers and allow them to evaluate more potential problems, reducing the risk of failure in the code.

### *Finding Defects*

Techniques to find defects can be divided into 3 categories:

---

<sup>8</sup> a type of software testing performed to identify bugs before releasing the product to real users or the public, it is done by testers.

<sup>9</sup> is performed by the real users of the product in a real (live) environment.



## Static Techniques

Testing is done without physically executing a program.

## Dynamic Techniques

This testing includes physically executing system components to identify defects.

An example of this would be executing a test case.

Software Test Cases are a specification of the inputs, execution conditions, testing procedure and expected results that define a single test to be executed to achieve a particular testing objective.

## Operational Techniques

This is where an operation system produces a deliverable containing a defect found by a user. The defect is found as a result of a failure.

Defects can be identified at any stage and by anyone in the development process. In order to ensure a quality product, a defect should be well documented. A report should contain:

- A summary
- Description
- Platform or build
- Steps to reproduce
- Expected result
- Actual result

## **Testing VS Debugging**

Testing is the process of looking for errors whilst debugging is the process of correcting the error.

### Black Box & White Box Testing

White Box Testing involves checking the way that things happen inside the system, AKA, looking at the code, meaning it is typically a static technique.

Black Box Testing on the other hand involves checking that the output is as expected based on the input.

## Example

### *Black Box*

Insert a coin into a drinks machine, and enter the drink code. If the drink is vended correctly the test was successful.

### *White Box*



Open the drinks machine up and check that the money went along the correct verification pathway, and the mixers for the drink were added in the expected sequence.

### Performance, Load & Stress Testing

Performance testing is a type of software testing that intends to determine how a system performs in terms of responsiveness and stability under a certain load. Load in this instance being how many transitions a system is performing in a given time.

Stress testing is a test where the load given to a system is at or beyond that which it is expected to be able to work with under normal conditions.

### Quality Assurance

There are other mechanisms outside of tests to ensure quality<sup>10</sup>. We will explore 4 below.

### Informal Reviews

These are often in the form of peer reviews, such as a developer reviewing the code produced by another developer. A quick and cheap type of review, that helps identify any obvious errors - two pairs of eyes are better than one. However, the review may also not pick up on some errors, as this review relies on humans; it is always open to human error.

### Walkthrough

In this, the product is walked through line by line or section by section to identify errors. Often particular scenarios are used to mimic live situations.

The review can be done by an individual or group, and is more likely to identify errors than an informal review particularly if it is done by a group.

### Technical Reviews

In this, the product is reviewed by a group of technical experts who usually use checklists of common problems to identify errors.

### Inspection

This is a formal meeting, with clearly defined roles and objectives. Roles include:

#### Inspectors

One or more inspectors review the product in detail.

#### Scribe

The scribe records all errors, actions and decisions.

---

<sup>10</sup> In this instance quality can be defined as a piece of software that meets functional and nonfunctional requirements.



## Moderator

The moderator is responsible for ensuring that the product is reviewed in a structured manner, conflicts are managed appropriately and all points of view are accounted for.

## **Implementation**

Is moving into the live environment. It takes careful planning, and is often conducted by software release engineers. There are many different implementation methods such as installing on a user's machine remotely, uploading to an app store, distributing via email and so on.

Sign-off occurs when the product is handed to the customer and they acknowledge receipt of the product and completion of the project. This usually includes signing a legally binding document, to ensure the customer cannot claim to not have agreed or received the product and the developing party may receive payment. Typically sign off occurs before deployment.

Post implementation reviews(PIR) are conducted at the end of a development. They evaluate the project as a whole, to determine what went well, what didn't and potential improvements for next time. They can include all members of the project team.

An ideal time to do a PIR is shortly after the project has been delivered, and when most of the problems have been addressed, as the project is still fresh in everyone's minds.

## **Maintenance**

The development lifecycle is not called a cycle for nothing. The development of a system doesn't end when it is released, but rather continues as the system is updated, faults are identified and fixed and improvements are made until, eventually, the system is decommissioned. This is the maintenance stage. Throughout this period, the system needs maintenance and support such as backup, patching, and audits.

Support is to help users such as providing training, tips and helping solve issues whilst maintenance is about keeping the software product working effectively.

Maintenance processes are owned by the IT department, and based upon predefined rules and tasks.

*Types Of Software Maintenance*

- Perfective
- Adaptive
- Corrective
- Preventive



### **Perfective Change**

50-55% of most maintenance work is attributed to perfective changes.

Perfective changes refers to the evolution of requirements and features to an existing system. This is because as a software gets exposed to users they will think of different ways to expand the system or suggest new features that they would like to see as part of the software, which in turn can become future enhancements to the system.

Perfective changes also include removing features from a system that are not effective or functional to the end goal of the system.

### **Adaptive Change**

20-25% of maintenance work is attributed to adaptive changes.

Adaptive change is triggered by changes in the environment the software lives in such as changes to the operating system, hardware, software dependencies or even organizational business rules and policies. These modifications to the environment can trigger a domino effect of changes.

### **Corrective**

20% of maintenance work is attributed to corrective changes.

Corrective changes address errors and faults in a software, most commonly, these changes are sprung by bug reports created by users. However, be mindful that sometimes problem reports submitted by users are actually enhancements of the system, not bugs.

### **Preventive Change**

5% of maintenance work is attributed to preventive changes.

Preventive changes refer to changes made focusing on decreasing the deterioration of a software in the long run. Restructuring and optimizing code and updating documentation are common preventive changes.

Executing preventive changes reduces the amount of unpredictable effects a software can have and helps it become scalable, stable, understandable and maintainable.

#### *Types Of Software Support*

### **1st Line Support**

An end user reports a problem, 1st line support technicians will log the issue and assign a unique reference number to the issue case. 1st line supporters may try to resolve the issue, however if they cannot they hand it over to 2nd line support.

### **2nd Line Support**

2nd line support are more technically knowledgeable, and will try to resolve the issue. If they cannot fix it, it is escalated to 3rd line support.

### **3rd Line Support**

3rd line supporters are typically developers who know a software system. They will almost always solve the issue.

## **Roles Of SDLC**

#### *Business Roles:*

### **Sponsor**

A person or organization that pays for or contributes to the cost of the project.

### **Senior Responsible Owner**

Accountable for a project meeting its objectives.

### **Business Analyst (BA)**





Someone who analyzes a business and documents its process and systems, assessing its business model and/or its integration with technology.

- Can be involved in all phases of the SDLC but main responsibilities are:
  - Requirements analysis, capture & tracking
  - Resource estimation and planning
  - Separation of functional and non-functional requirements

### **Domain Expert**

A person who is an authority/very knowledgeable in a particular area/topic.

### **End User**

A person who actually uses a particular product or service.

### *Project Roles:*

#### **Project Manager**

A role dedicated to the planning, scheduling, resource allocation, execution, tracking and delivery of a software product.

- Can be involved in all stages of SDLC
- They are the single point of contact (poc) for development teams and the client
- Ensure SDLC standards are upheld, such as ISO or CMMI
  - The International Standards Organization (ISO) and Capability Maturity Model Integration (CMMI) are two standard ratings and guidelines given to organizations that develop software

#### **Team Leader**

Someone who provides guidance, instruction, direction and leadership.

#### **Work Package Manager**

A work package is a group of related tasks within a project, they are often thought of as sub-projects within a larger one. Work packages are the smallest unit of work that a project can be broken down into. A Package Manager manages these 'sub-projects'.

### *Technical Roles:*

#### **Technical Architect**

Responsible for defining the overall structure of a program or system. They act as a project manager, overseeing IT assignments that are aimed at improving the business and ensuring the project runs smoothly.

### **Designer**

Is responsible for designing a software model that fulfils the specifications, usually by using diagramming tools. They are also responsible for documenting the design and choosing the system architecture.

### **Solution Developer**

Creates the computer products needed in order to accomplish particular goals, ranging from custom software to graphical design. Customer needs must be determined by constant communication.

### **Solution Tester**

Tester of above. They will perform and record testing (black-box and white-box testing), and be responsible for quality assurance by using static and dynamic analysis tools such as walkthroughs, automated scripts and performance tools.

### *Implementation And Support Roles:*

#### **Release Manager**

Release management is the process of managing, planning, scheduling and controlling a software build through different stages and environments, including testing and deployment.

#### **Database Administrator**

This role may include: capacity planning, installation and configuration, database design, migration, security as well as backup and data recovery.

#### **System Administration**

Is responsible for upkeep, configuration and reliable operation of computer systems.

### *Team Working & Structure*

Effective team working requires:

- Communication
- Composition

Team composition involves putting together the right set of individuals with relevant expertise to accomplish the team goals and tasks, to maximize team effectiveness

- Maturity

Maturity is not just about age, but how someone is mindful of others before they say or act

## Deliverables Of SDLC

Many deliverables are a way of detailing what is understood about the system in terms of what it needs to do, how it should do it and how it gets delivered.

Deliverables include:

- Requirement Documents

(PRD) is a document containing all of the requirements of a product. It is written to give an understanding of what a product should do.

- Class/Entity Relationship Models

- [View here](#)

- Use Case Diagrams

- [View here](#)

- Process Models ei V-model/Waterfall

- [View here](#)

- Component Diagrams

- [View here](#)

- State Transition Diagrams

- [View here](#)

- Sequence Diagrams

- [View here](#)

- Activity Diagrams

- [View here](#)

- Data Flow Diagrams

- [View here](#)

- Test Plans

- Test Scripts

- Implementation Plans

- System Components And Working Software



| Stage                    | Deliverables  | People Involved in Stage  |
|--------------------------|---|---|
| Requirements Engineering | <ul style="list-style-type: none"> <li>- Business Requirement Documentation (Brd)</li> <li>- Software Requirement Specifications (Srs)</li> <li>- Technical Requirement Specifications</li> </ul> | Top Level Management Of The Business Are Involved In Gathering The Requirements Of The Software: <ul style="list-style-type: none"> <li>- Project Managers</li> <li>- Directors</li> <li>- Sales, Marketing &amp; Consulting</li> <li>- Stakeholders</li> <li>- Business Analyst</li> </ul> |
| Design                   | <ul style="list-style-type: none"> <li>- Use Case Diagram</li> <li>- Class Diagram</li> <li>- Entity Relationship Diagram</li> <li>- Component Diagram</li> </ul>                                 | <ul style="list-style-type: none"> <li>- Designers</li> <li>- Business Analysts</li> <li>- Architects</li> </ul>  |
| Development              | <ul style="list-style-type: none"> <li>- Functional Code</li> </ul>   | <ul style="list-style-type: none"> <li>- Developers</li> </ul>  |
| Testing                  | <ul style="list-style-type: none"> <li>- Test Plans</li> <li>- Test Scripts</li> </ul>  | <ul style="list-style-type: none"> <li>- Testers</li> </ul>   |
| Implementation           | <ul style="list-style-type: none"> <li>- Working Product</li> </ul>   | <ul style="list-style-type: none"> <li>- Release Managers</li> <li>- Developers</li> </ul>  |

### Diagram Deliverables

A model is a representation of a subject, it can be a real-world object, imagined subjects (such as a video game spaceship) or a system that either already exists or may exist in the future.

There are many modelling notations and frameworks in existence, but the following are ones you need to be aware of for the BCS Software Development Essentials exam.

The UML models below can be split further into 2 types:

| Behavioural Diagrams             | Structural Diagrams |
|----------------------------------|---------------------|
| Use Case                         | Class / Entity      |
| Sequence                         | Component           |
| State Machine (State Transition) |                     |
| Activity Diagram                 |                     |

## Brief History Of UML

The first object oriented programming language was introduced in 1965 called Simula, with many more emerging afterwards. With these, came numerous modelling languages to allow for analysis and design. Naturally, not all survived and 3 dominant methods emerged.

### Object-oriented Analysis & Design (Ooad) – By Grady Booch

Complex, and consisted of lots of diagrams, good for low-level design and fine detail, but weaker at analysis.

### The Object Modeling Technique (Omt) – By Jim Rumbaugh

Consisted of simpler diagrams, allowing for mid to high-level design and easier analysis.

### The Object-oriented Software Engineering Method (Oose) – By Ivar Jacobson

Who's big feature was 'Use Classes', which allowed for modelling on how a system interacts with users. Good at high-level design.

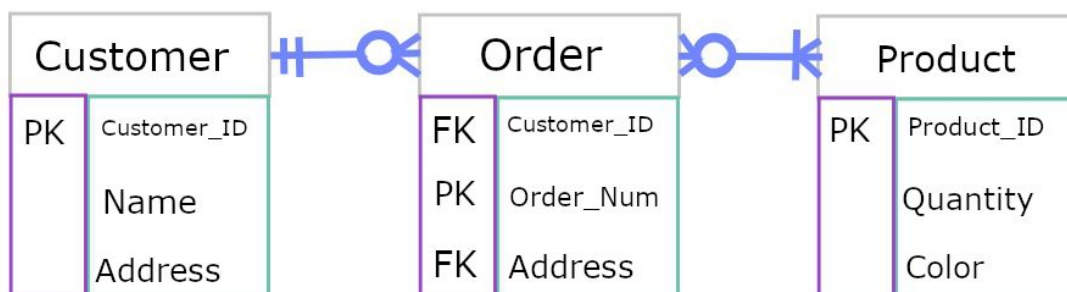
In 1994 Booch & Rumbaugh joined forces, and soon Jacobson joined too and UML was born. In 1997 UML 1.0 was introduced.

## Entity Relationship Models

ERM is a theoretical and conceptual way of showing data relationships. Most often used to design or debug relational databases.

ER diagrams are composed of entities, relationships (cardinality- which defines relationships in terms of numbers) and attributes.

If you've ever worked with Databases and/or SQL you will find this model very easy to understand.



## Entities



A definable thing such as a person, object or event that can have data stored about it. Think of them as nouns<sup>11</sup>, such as a customer or product. They are shown as rectangles.



Each entity (type) represents an individual table in a database. And in the database, the entities will be the rows whilst the attributes are the columns.

## Entity Categories

The available categories for entities are strong, weak or associative.

A strong entity can be defined solely by its own attributes, whilst a weak cannot - it is an entity that depends on other entities as it doesn't have any key attribute of its own.

Associative associates entities within an entity set.

## Entity Type & Set

An entity type is a group of definable things such as customers or students whilst an entity is a specific student or customer.

An entity set is the same as entity type, but defined at a particular point in time. Such as customers who purchased in the last 3 months.

## Entity Keys (Optional)

*Highlighted in the diagram example above as purple.*

Entity keys refer to an attribute that uniquely defines an entity.

- Primary Keys (*shown above as PK*)
  - A candidate key chosen by the database designer to uniquely identify the entity
- Foreign Keys (*shown above as FK*)
  - Identifies the relationship between entities - its a term used for an attribute that is the primary key of another table
- Super Keys
  - A attribute or set of attributes that uniquely identifies an entity - there can be many of these.
  - In layman terms this is any combination of attributes that uniquely identify a row/entity where no combination is the same.
- Candidate Keys
  - Otherwise known as a minimal super key, it has the least possible number of attributes to still be a super key

---

<sup>11</sup> A word used to identify any of a class of people, places or things.

## Super & Candidate Key Example

| <b><i>book_ID</i></b> | <b><i>name</i></b>    | <b><i>author</i></b> |
|-----------------------|-----------------------|----------------------|
| 1                     | Learn CSS             | Chairbear            |
| 2                     | Learn JavaScript      | Chairbear            |
| 3                     | Learn HTML            | Chairbear            |
| 4                     | Learn Web Development | Chairbear            |

The combination of *name* and *author* is a superkey, as none of the 2 value combinations are the same.

[Learn CSS, Chairbear] is different from [Learn JavaScript, Chairbear]

| <b><i>name</i></b> | <b><i>author</i></b> |
|--------------------|----------------------|
| Learn CSS          | Chairbear            |
| Learn JavaScript   | Chairbear            |

If ID 4 had the *name* of Chairbear and *author* of Chairbear, this would then not be a unique combination as both attributes have the same value, and so name and author would then not be super keys.

| <b><i>book_ID</i></b> | <b><i>name</i></b> | <b><i>author</i></b> |
|-----------------------|--------------------|----------------------|
| 3                     | Learn HTML         | Chairbear            |
| 4                     | Chairbear          | Chairbear            |

Going back to our original table, *book\_ID* is a superkey as its values are unique - no two values combined are the same.

| <b><i>book_ID</i></b> |
|-----------------------|
| 1                     |
| 2                     |
| 3                     |
| 4                     |



If we say no two combinations of *name,book\_ID* and *author* are the same, we can say this is a super key.

[1,Learn CSS, Chairbear] is different from [2,Learn JavaScript,Chairbear]

However, when working with relational databases, if you combine all of the attributes like above they will of course make a super key. As no two rows in a database table will be exactly the same. This is called redundancy. Some super keys, like the one above, will have redundancies.

A candidate key is a super key without redundancy, and it is not further reducible. We can see that *book\_ID* is unique throughout, and so there is no reason to say that the combination of *name,book\_ID* and *author* is unique, as *name* and *author* are made redundant in that combination. *book\_ID* is a candidate key.

## Attributes

A property or characteristic of an entity. Shown as an oval or circle.



## Attributes Categories

The available categories are simple, composite, derived, multi and single.

A simple attribute is an attribute value that is atomic, and cannot be further divided. Such as a phone number.

Composite, is a sub attribute composed of many other attributes. For example, an address attribute could be composed from street, zipcode and country attributes. A composite attribute is represented by an oval with other ovals branching from it (behind it).

Derived attributes are calculated/derived from another attribute such as age from birthdate.

Multi value attributes, are attributes where the value is or can be multiple, such as multiple phone numbers for one person.

Single value attributes contain just one value.

## Cardinality And Relationships

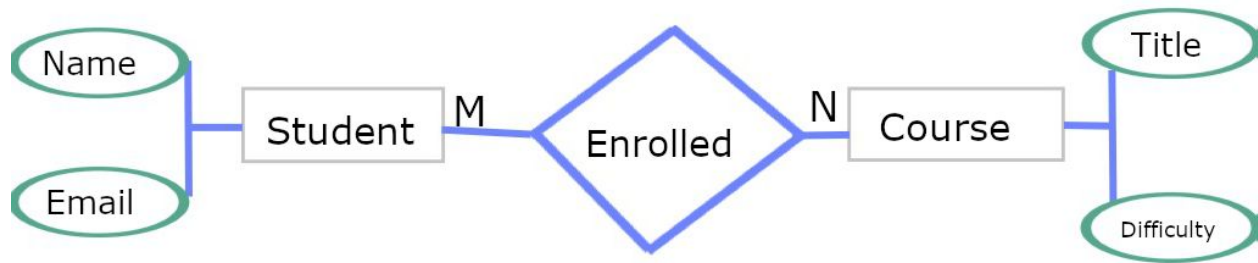
Relationships are how entities act upon each other/ are associated. Think of them as verbs<sup>12</sup>.

---

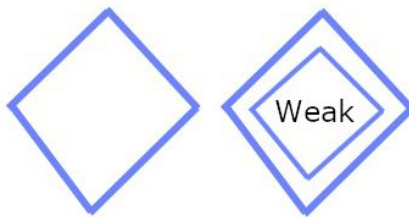
<sup>12</sup> A word used to describe an action, state or occurrence.



For example, a student enrolling for a course. The 2 entities are the course and the student, and the relationship would be the act of enrolling.



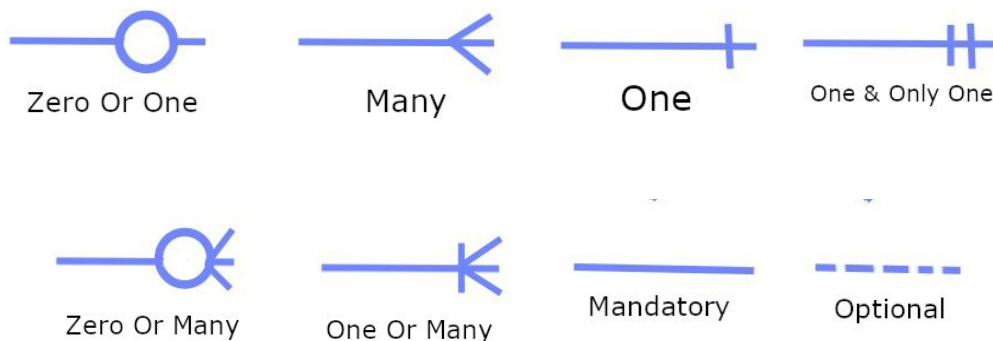
Relationships are shown as diamonds or labels on connecting lines.



A recursive relationship is where the same entity participates more than once in a relationship.

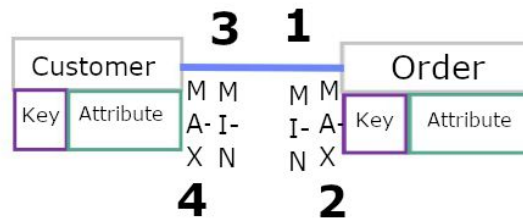
Cardinality defines the numerical value of the relationship between entities. The main cardinalities you'll see are:

- One To One
  - For example, 1 student to 1 email address
- One To Many/ Many To One
  - For example, 1 student enrolled in multiple courses
- Many To Many
  - For example, students as a group are associated with multiple teachers



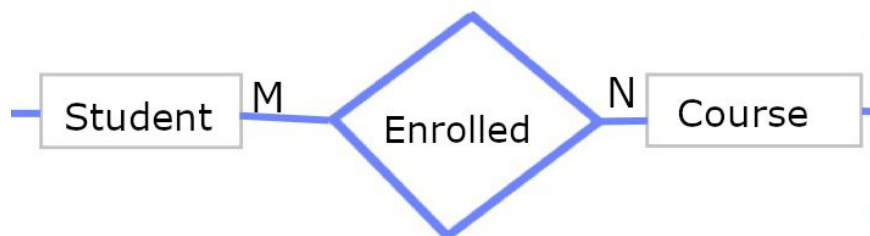
A mandatory relationship will be shown with a solid line, whilst an optional relationship will have a dotted line.

### First Example Explained - Working Out Cardinality



- 1) What is the MINIMUM number of orders a customer can have?
  - a) A customer can exist without any orders. The answer is 0.
- 2) What is the MAXIMUM number of orders a customer can have?
  - a) A customer can exist with many (infinite) orders. The answer is many.
- 3) What is the MINIMUM number of customers an order can have?
  - a) An order needs a minimum of 1 customer to exist. The answer is 1.
- 4) What is the MAXIMUM number of customers an order can have?
  - a) An order can only have 1 customer, as one order having multiple customers would be confusing. The answer is 1.

## Second Example Explained



The M and N represent any integer<sup>13</sup>, meaning any number of students can study any number of courses. M&N are variables. If it were M --- 1 it would suggest a many to one relationship.

## Class Relationship Models

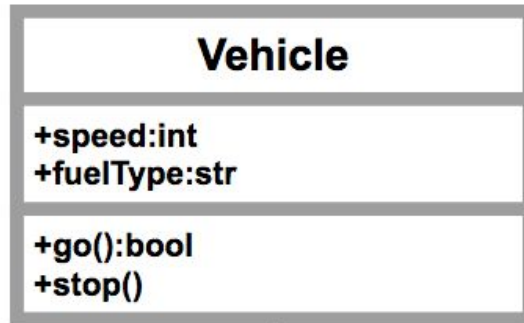
A class relationship model is a static structure diagram that represents the structure of a system through showing its classes, their attributes and methods, and the relationship between them.

A class, in this instance then, is a group of objects all with similar roles in the system.

In class relationship models, the class shape is a rectangle with three rows.

<sup>13</sup> A whole number





The top row contains the name of the class. This section is always required.

The middle row contains the attributes of the class, each attribute takes up a line. The attribute type is shown after the colon. Attributes (also known as fields, properties or variables) map onto member variables<sup>14</sup> in code.

The bottom section expresses the methods (also called operations) that the class may use, and each operation takes up its own line. The operations describe how a class interacts with data, and map onto class methods in code.

Similar to attributes, the return type of a method is shown after a colon. And if the method contains parameters within its parentheses, the type will be after the parameter, separated by a colon.

### Member Access Modifiers

Classes can have different access levels depending on the access modifier, otherwise known as its visibility. The visibility of an attribute or method sets the accessibility of such.

#### Public (+)

An attribute or method is public if the + (plus) sign is before it. This means it can be accessed by another class.

#### Private (-)

An attribute or method is private if the - (minus) sign is before it. This means they cannot be accessed by any other class or subclass.

---

<sup>14</sup> In object-oriented programming, a member variable is a variable that is associated with a specific object, and accessible for all its methods. For example (This is a Java Example):

```
public class IntegerClass {
    int anInteger;
}
```

The above declares an *integer member variable* named `anInteger` within the class, `IntegerClass`.

Resources: <http://journals.ecs.soton.ac.uk/java/tutorial/java/javaOO/variables.html>  
[https://en.wikipedia.org/wiki/Member\\_variable](https://en.wikipedia.org/wiki/Member_variable)



## Protected (#)

An attribute or method is protected if the # (hash) sign is before it. These can only be accessed by the same class or it's subclasses

## Package (~)

If the ~ (tilde) sign is before an attribute or method, this means the visibility is set so they can be used by any other class as long as it's in the same package

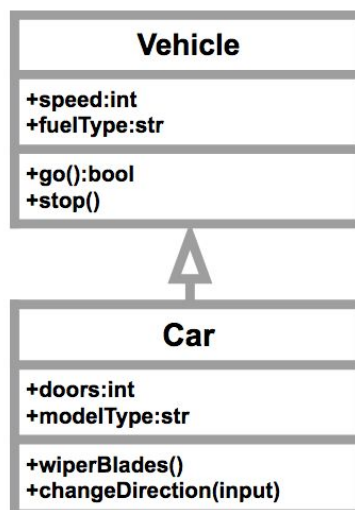
## Relationships

A class may be involved in one or more relationships with other classes. There are 5 types of relationship: inheritance, simple association, aggregation, composition and dependency.

### Inheritance (also known as generalization)

Is the process of a child (or sub-class) taking on the functionality of a parent (or superclass)

They are represented with a solid line with a hollow arrowhead that points from the child to the parent class.



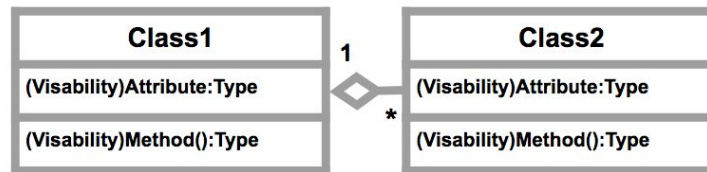
In the above example, the 'car' object would inherit all of the attributes (speed and fuelType) and methods (go() and stop()) of the parent class 'vehicle', in addition to the specific attributes and methods of its own class (such as doors, or changeDirection()).

## Association

In association, both classes are aware of each other and their relationship. There is no dependency, it's just a basic association which is depicted by a simple line.

## Aggregation

This is a type of association that specifies a whole and its parts - if a part can exist outside of a whole it is an aggregation relationship. This is represented by an open diamond. The unfilled diamond is at the association end.

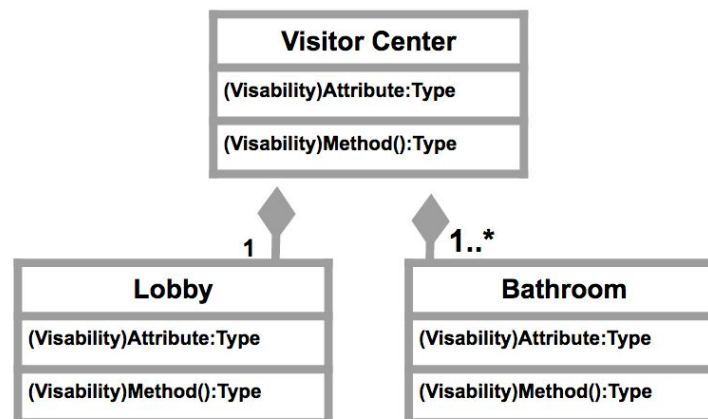


### Example Explained

Class2 is a part of class1. Many instances of class2 can be associated with class1, however objects of class1 and 2 have separate lifetimes.

### Composition

A special type of aggregation where parts are destroyed when the whole is destroyed. Depicted by a solid line with a filled diamond at the association end.

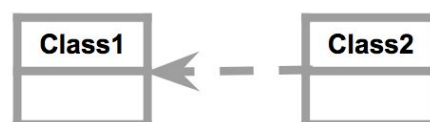


### Example Explained

If the visitor centre is torn down, the lobby and bathrooms would also cease to exist.

### Dependency

This exists between two classes if changes to the definition of one may cause changes to the other, but not visa versa. Represented by a dashed line with an arrow, from dependent to independent.



### Example Explained

Class2 is dependent on class1.

## Relationship Names

Written in the middle of the association line. Good names make sense when you read them outloud.



## Example Explained

Every spreadsheet contains some number of cells.

## Multiplicity

| Symbol | Definition   |
|--------|--|
| 0..1   | Zero To One  |
| n      | A Specific Amount (in the visitor example, it uses 1 for the lobby meaning it can only have one lobby, but can have one or many bathrooms) |
| 0..*   | Zero To Many   |
| 1..*   | One To Many  |
| M..N   | A Specific Number Range  |

## Use Case Diagrams

In software development, a use case<sup>15</sup> is a list of actions or event steps typically defining the interactions between a role (in the Unified Modeling Language(UML) this is called an actor) and a system to achieve a goal. The actor can be human or an external system.

They are used to display interactions between users and a system at a high level.

Common components in a use case include:

---

<sup>15</sup> Use cases are textual descriptions of the functionality of a system from the users' perspective whilst use case diagrams are used to visually depict the functionality that the system will provide, and which users will communicate with the system when it provides said functionality.

- **Actors**
  - The user that interacts with a system. They must be external objects that produce or consume data
- **System**
  - A specific sequence of actions and interactions between actors and the system. AKA a scenario
- **Goals**
  - The end result of a use case. A successful diagram should describe activities used to reach the goal

## **Use Case Notation**

### **Use Cases**

These are horizontal ovals that represent the different uses a user may have with a system.

### **Actors**

These are represented by stick figures, that represent the people employing the use case. Always drawn outside of the system boundary box. There are two types of actors, primary and secondary.

Primary actors initiate the use of a system, and are always on the left side of a system boundary box. Secondary actors are reactionary, and always on the right side.

### **Associations**

Associations are represented with lines between actors and use cases.

### **System Boundary Boxes**

A box that sets a system scope to use cases. All use cases outside a box would be considered outside the scope for that system.

### **Include Relationships**

Shows a dependency between a base use case and an include case. Both cases will be executed. Think of it as a base case requiring an include case in order to be complete. IRs are depicted by a dashed line, with an arrow that points at the include case, with 'include' written between double chevrons.

### **Extends Relationships**

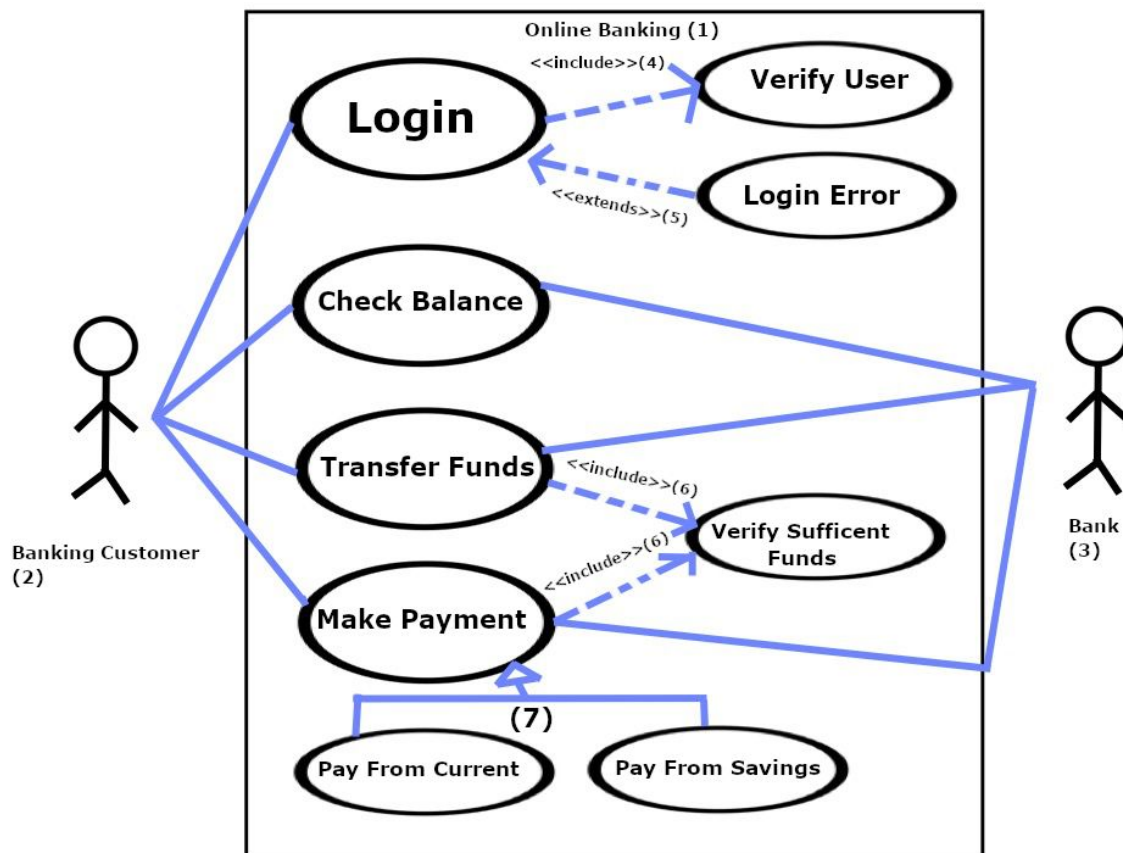
When a base use case is executed, the extend use case will happen only when certain criteria are met. ERs are depicted by a dashed line, with an arrow that points at the base case, with 'extends' written between double chevrons.

### **Generalisation Relationships**

Specialised uses cases point at a generalised use case. They represent more options or categories beneath the general case. And are depicted with solid lines and hollow arrows.

## Packages

A UML shape that allows you to put different elements into groups, these groupings are represented as file folders.



- 1) Name of system at the top
- 2) Primary Actors
- 3) Secondary Actors
- 4) Include Relationship
- 5) Extend Relationship
- 6) Multiple base use cases can point at the same include case
- 7) Generalisation Relationship

## Use Case Descriptions

As well as producing a use case diagram to show the use cases in a system, each use case can be described individually. This is where use case descriptions come in.



|                           |  |
|---------------------------|--|
| <b>Name</b>               |  |
| <b>Actors</b>             |  |
| <b>Trigger</b>            |  |
| <b>Preconditions</b>      |  |
| <b>Post Conditions</b>    |  |
| <b>Success Scenario</b>   |  |
| <b>Alternatives Flows</b> |  |

Generic format displayed above.

### Triggers

What starts the use-case? For example, a customer reports a claim, or customer inserts card.

### Preconditions

What the system needs to be true before running the use-case. For example, a user account exists, or user has enough money in their account.

### Post Conditions

Are the outcomes of the use-case. For example, money was transferred to the user account, or user is logged in.

### Success Scenario

Is the main story-line of the use-case.


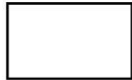



It is written under the assumption that everything is okay (no errors occur), and it leads directly to the desired outcome of the use-case. A success scenario is composed of a sequence of action steps.


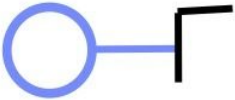

### Alternate Flows

This section should contain any possible other ways in which the use case might turn out.

### *Component Diagrams*

Are more specialised versions of the Class Diagram, in which it breaks a system down into smaller components and visualises the relationships between.

| Symbol   | Symbol Definition   |
|--|---|
|   | <p><b>Component</b></p> <p>An entity required to execute a function. A provides and consumes behaviour through interfaces and/or other components.</p>        |
|   | <p><b>Node</b></p> <p>This represents hardware or software objects which are of a higher level than components.</p>   |
| <div> <div> &lt;&lt;Interface&gt;&gt;<br/>Interface </div> <div> +Attribute1:Type =<br/>Default Value<br/>-Attribute2:Type<br/>-Attribute:Type </div> </div> <div> <div> &lt;&lt;Component&gt;&gt;<br/>Name </div> <div> &lt;&lt;Provided Interfaces&gt;&gt;<br/>Interface Name<br/>&lt;&lt;Required Interfaces&gt;&gt;<br/>Interface Name </div> </div> | <p><b>Interface</b></p> <p>Shows input or materials that a component either receives or provides. Interfaces can be represented by text notes or symbols.</p> |
|   | <p><b>Port</b></p> <p>Specifies a separate interaction point between the component and the environment.</p>   |
|   | <p><b>Package</b></p> <p>This is a group together of multiple elements of the system.</p>   |
|   | <p><b>Note</b></p> <p>Allows Developers to add a note(additional info) to the diagram.</p>  |

|   |  |
|---|--|
|  | Dependency   |
|  | <b>Provided Interface</b><br>Depicted as a straight line from the component box with an attached circle, this symbol represents the interface where a component produces information used by the required interface of another component.              |
|  | <b>Required Interface</b><br>Depicted as a straight line from the component box with a half circle, or dashed line with an open arrow, these symbols represent the interfaces where a component requires information in order to perform its function. |

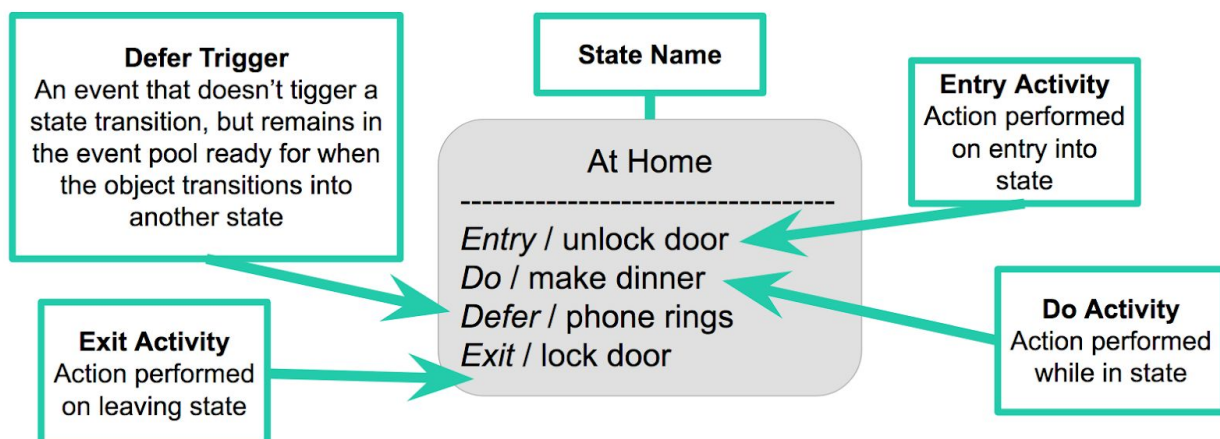
### State Transition Diagrams

State transition diagrams - also known as state machine diagrams - are used to describe all of the states that an object can have. This includes the events under which an object changes state (AKA *transitions* to a new state), the conditions that must be fulfilled before the transition will occur (called guards, explored further below), and the activities undertaken during the life of an object.

### Notation

#### State

This is a condition during the lifecycle of an object in which some condition is satisfied, an action is performed, or waiting on events occurs. Depicted as a rectangle with rounded corners.



## First State

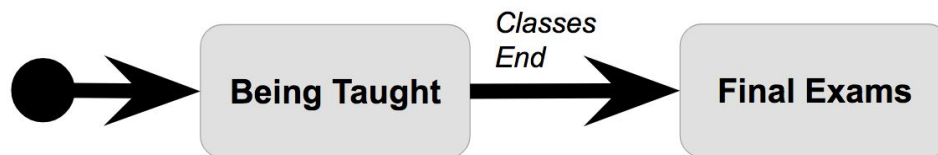
This is a marker for the first state in the process, and is depicted by a dark (filled) circle with a transition arrow.

## Events & Triggers

This is an occurrence that may trigger a state transition. There are 4 event types:

- A signal from outside the system
- An invocation from inside the system
- The passage of time
- A condition becoming true

They are labeled above transition arrows.



For example, 'classes end' is the event here that triggers the end of the being taught state and the beginning of the final exams state.

## Guard

A boolean expression which when true enables an event to cause a transition. These are written above the transition arrow.

## Choice Pseudostate

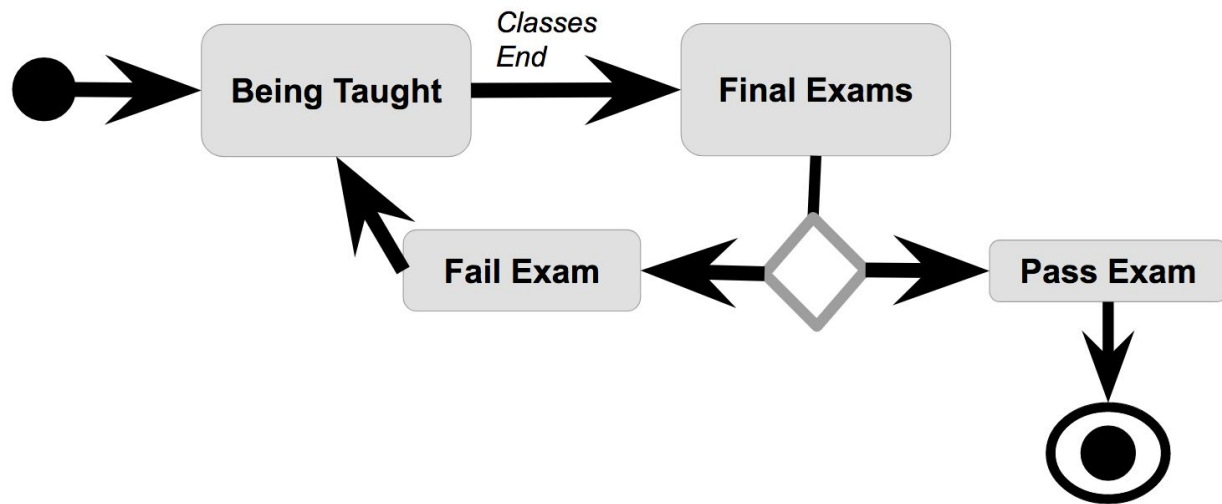
Represented by a hollow diamond, these represent a dynamic condition with branched potential results.

## Exit Point

The point at which an object escapes the state machine, it is typically used if the process is not completed but had to be escaped due to an error. Depicted by a circle with an X through it.

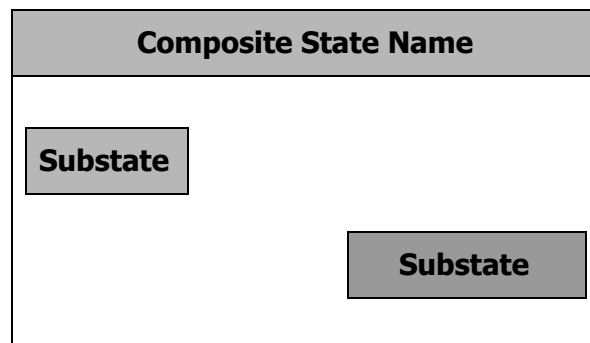
## Terminator

A circle with a dot within it, it indicates that a process is terminated.



### Substates & Composite States

A composite state is a state that has substates nested into it. Often depicted as an encompassing rectangle, with the name of the composite state highlighted within its own section.






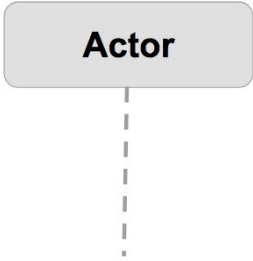

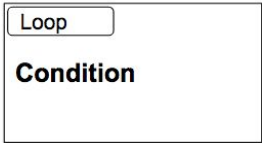
A substrate, then, is a state contained within a composite state's region.

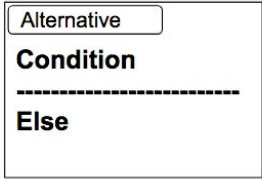



### Sequence Diagrams

Sequence diagrams are interaction diagrams that detail how operations are carried out. They depict high-level interactions between users and the system, or between the system and other systems. Sequence diagrams are sometimes known as event diagrams or event scenarios.

They show how elements interact over time, organized according to object (horizontally) and time (vertically).

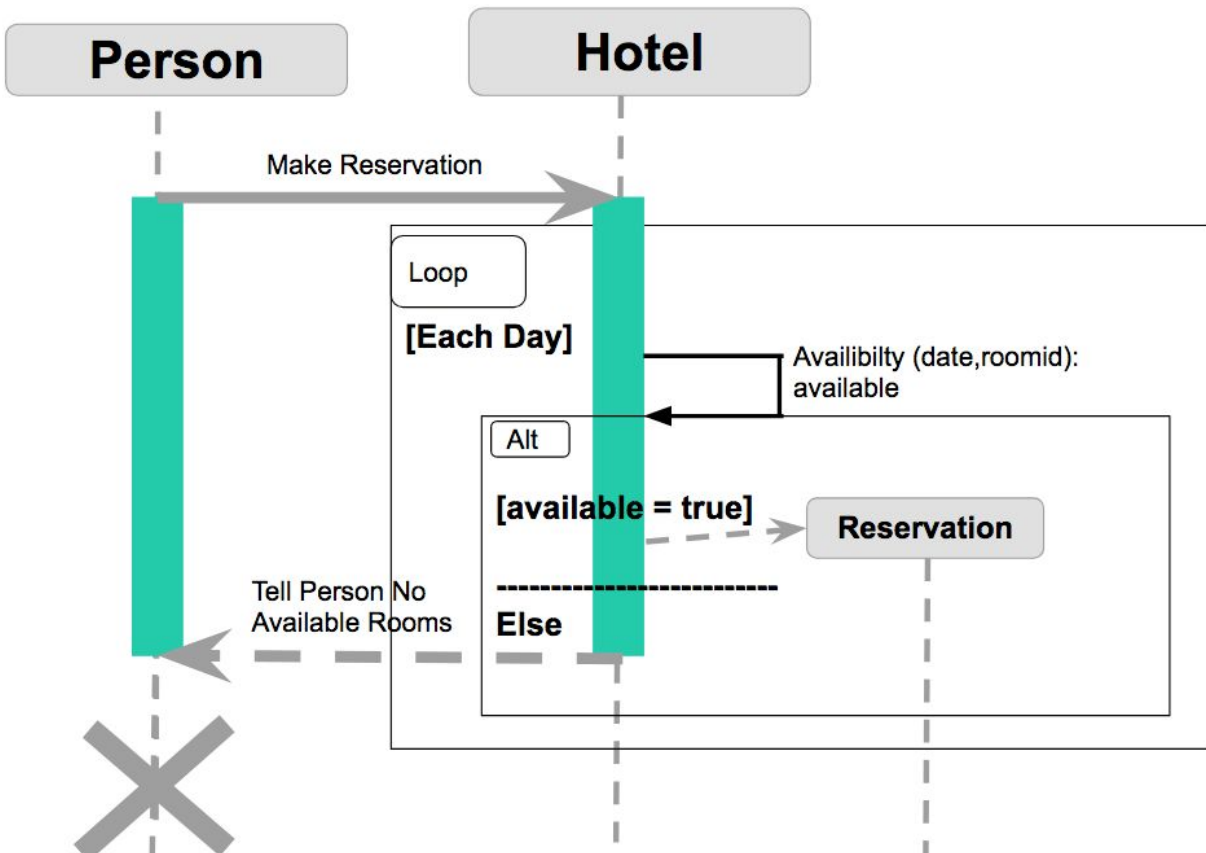
| Notation Symbol | Notation Meaning |
|-----------------|------------------|
|-----------------|------------------|

|   |   |
|---|---|
|    | <p><b>Object Symbol</b><br/>Represents a class or object, and demonstrates how an object will behave in the context of the system. Class attributes should not be listed in this.</p> |
|    | <p><b>Activation Symbol</b><br/>Represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.</p>                |
|    | <p><b>Actor</b><br/>Entities that interact with, or are external to the system.</p>   |
|  | <p><b>Lifeline</b><br/>Represents the passage of time as it extends downward. Lifelines may begin with a labeled rectangle shape or an actor symbol.</p>                              |
|  | <p><b>Delete Message</b><br/>This message destroys an object. It is depicted by a solid line with an X at the end.</p>  |
|  | <p><b>Option Loop</b><br/>Used to model if/then scenarios. Used for circumstances that will only occur under certain conditions.</p>  |

|  |  |
|--|--|
|  <p>The diagram shows a rectangular box representing an alternative message. It has a small tab at the top left labeled "Alternative". Below the tab, the word "Condition" is followed by a dashed horizontal line, and below that, the word "Else" is shown.</p> | <p><b>Alternative</b><br/>Symbolizes a choice between two or more message sequences.</p>   |
|  <p>The diagram shows a solid horizontal arrow pointing to the right, representing a synchronous message.</p>   | <p><b>Synchronous Message</b><br/>Represented by a solid line with a solid arrowhead. This is used when a sender must wait for a response to a message before it continues.</p>  |
|  <p>The diagram shows a solid horizontal arrow pointing to the right, representing an asynchronous message.</p>   | <p><b>Asynchronous Message</b><br/>Represented by a solid line with a lined arrowhead. These don't require a response before the sender continues.</p> <p>Unlike synchronous messages, only the call should be included in the diagram, not the reply too.</p> |
|  <p>The diagram shows a dashed horizontal arrow pointing to the left, representing a reply message.</p>   | <p><b>Reply Message</b><br/>Represented by a dashed line with a lined arrowhead, these messages are replies to calls.</p>  |

### Basic Example

This example includes a Self Message which is a kind of message that represents the invocation of a message of the same lifeline.



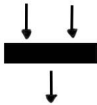

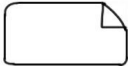







### Activity Diagrams

Activity diagrams are very similar to state transition diagrams. They are a type of flowchart used in object oriented (OO) programming analysis and design to describe a business process or workflow. They support parallel (concurrent) tasks which a pure flowchart cannot handle.

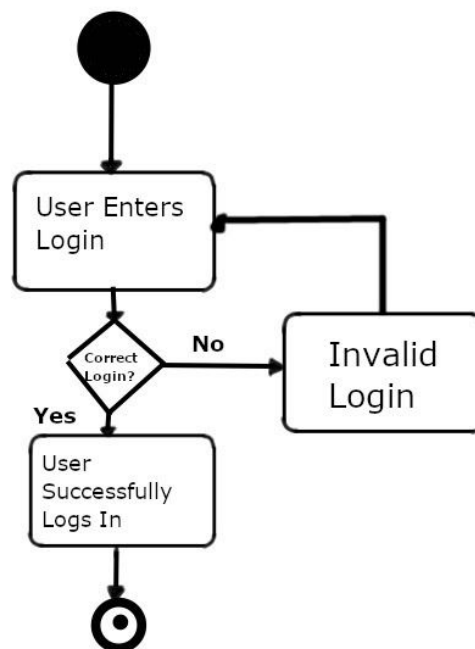
| Notation Symbol | Notation Meaning  |
|-----------------|---|
|                 | Start Symbol<br>Represents the beginning of a process, and can be used by itself or with a note that explains the starting point.   |
|                 | Activity Symbol<br>Indicated the activities that make up a modeled process, with short descriptions within. Can often be used to indicate operations (methods) that a software should have. |



|   |  |
|---|--|
|    | <p>Connector<br/>Shows the directional flow</p>  |
|    | <p>Fork<br/>Splits a single activity flow into two concurrent activities.</p> <p>Also known as concurrent (parallel) flow symbols.</p>   |
|    | <p>Join Bar<br/>Combines two concurrent activities and reintroduces them to a flow.</p> <p>Also known as concurrent (parallel) flow symbols.</p>   |
|    | <p>Decision<br/>Always has at least two paths branching out from it with condition text to allow users to view options.</p> <p>This can also be used to show a merge. Through which at least two arrows enter the diamond, and one leaves.</p> |
|  | <p>Note<br/>Used to communicate messages that don't fit within the diagram itself.</p>   |
|  | <p>Send Symbol<br/>Indicates that a signal is being sent to an activity.</p>   |
|  | <p>Receive Symbol<br/>Demonstrates acceptance of the event.</p>  |
|  | <p>Shallow History Pseudostate<br/>Used to represent a transition that invokes the last active state.</p>  |

|   |   |
|---|---|
|  | Flow Stop<br>Represents the end of a specific process flow                        |
|  | End<br>Marks the end of an activity and the completion of all flows in a process. |

### Basic Example



## Software Execution

A program is a set of instructions to tell a computer what to do. They are written using a programming language such as C#. There are three major levels of language in programming, these are high-level, assembly, and machine (low-level) language.

### High-Level Languages

High level code is that which developers use. It is code similar to written languages so that it is easier for humans to comprehend. High-level languages include C#, C++, Java, PHP, ect.

```

Console.WriteLine("What is your age?");

int age = Convert.ToInt32(Console.ReadLine());

if (age < 18){

    Console.WriteLine("Your too young for this site!");

} else

{

    Console.WriteLine("Welcome!");

}

```

All of these languages must be converted to machine language before the CPU<sup>16</sup> can process the program.

### *Translators*

The term translator is used for software that converts programming languages code into machine level code for a CPU to execute. For high-level languages, the translation is done by either a compiler or an interpreter. For assembly languages, the translation is done by the assembler.



### Pre-Processor

---

<sup>16</sup> Central processing unit - is the electronic circuitry within a computer that executes programs.

The pre-processor performs some formatting of the source code before it is handed to the compiler. It 'tidies-up' code to ensure that it is in the correct format for the compiler to work with. An example of the processes the pre-processor does below:

- Concatenate source code that is split over multiple lines.
- Remove any comments and replace them with a space.
- Copy any code in from 'included' libraries or files.
- Process any escape sequences.

After the file is pre-processed it is then passed to the compiler.

### Compiler

The compiler translates the whole program into machine code before the program is run.

The output machine code is saved and stored separately as object code. For example, in different DLLs<sup>17</sup>.

Compilation is often slow, but the resulting machine code can be executed quickly.

| Benefits  | Drawbacks   |
|---|---|
| Once the source code has been compiled, the compiler and or source code is no longer needed   | Object code will only run on computers that have the same platform  |
| Increased security and privacy, as well as protection of copyright material. As people will find it hard to figure out what the original source code is from the object code. The process of discovering this is called reverse engineering | It can take a long time to debug, because the whole program has to be converted from source code to object code every time a alteration is made to the code |

### Interpreter

Interpreters are often used for web applications, as they work by translating one statement of the code at a time – typically only the code that is required. For example, only one route through an if statement.

| Benefits                             | Drawbacks                                    |
|--------------------------------------|--|
| You do not need to compile the whole | The source code must be distributed to other |

---

<sup>17</sup> Is a library that contains code and data that can be used by more than one program at the same time.



|   |   |
|---|---|
| program in order to run sections of code  |   |
| As code is translated each time it is executed, program code can be run on processors with different instruction sets | The source code can only be translated and executed on a computer that has the same interpreter   |
|   | No matter how many times a section of code is revisited in the same program, it will need translating every time. This increases the time needed to execute a program |

### Linker

A linker combines the object files from the compiler into a single executable file.

## Assembly Language

Whilst programs are in machine code, programmers devised assembly languages which are short programmed commands, to execute particular instructions from a CPU's instruction set. It gives high levels of control due to the direct manipulation of the CPU.

Each command, such as ADD, SUB, MPY, DIV, LOAD and STOR, are called operation code (often shortened to OpCode). And they usually consist of 1 to 4 characters, and are often used alongside operands (which are the actual data).

As aforementioned, when assembly code is executed the CPU uses an assembler to translate the code into binary (MLC). Assembly code is not very portable, as it is based on one processor type.

## Machine Language

Machine language is a low-level programming language used to directly control a computer's central processing unit. Binary is an example of machine language.

### *Analogue VS Digital*

And whilst computers deal with binary data, much of the data in real life is not in binary form. Analogy data is continuous, for example sound waves. Analogue data regularly has to be converted into a digital form so that it can be processed and stored by computers. To do this, measurements (samples) are taken of the analogue data and these values stored as approximate representations.

The greater the sample size, and frequency, the closer the representation is to the source. An example of this is using microphones. You can record your voice and the analogue data will be

saved on your computer as a digital conversion. Another example is the reverse of analogue to digital, and instead digital to analog. A sound card in a PC includes a DAC (digital to analog convertor), this is how video game music will be translated from digital to sound waves.

# Introduction To C#

## What is C#?

C# is part of the .NET Framework which belongs to Microsoft. The .NET is an Execution Environment automatically included in Windows, which includes reusable class libraries (Framework Class Library - FCL) and language compilers. The virtual execution environment is called the Common Language Runtime (CLR) and it allows the language and class libraries to work together.

All the .NET languages (C#, F# and Visual Basic) are compiled down to a Common Intermediate Language (CIL), meaning that assemblies from either language will work with the other .NET languages.

C# is an object oriented programming language, this means that it creates objects that hold both data and methods. Unlike procedural programming that focuses on writing procedures or methods that perform operations on data. With procedural programming, code is executed sequentially ( from top to bottom, line after line), and separate code modules (methods, functions, procedures and subroutines) are created and called from the main code. This is done to break the code into smaller manageable chunks.

C# is case sensitive, and called a 'strongly' typed language.

## Creating Variables

Variables are containers for storing data. They must be declared in order to allocate memory space. There are two ways to declare variables in C#, below we will explore both methods and their syntax.

### Var & Const Method

Const works similarly to var (which is short for variable), in which you use the keyword var, or const followed by a name for the variable. Then an equal sign (the assignment operator), and a value you want to put into the variable.

Const is short for constant, meaning that the value of the variable should not change. It lets you protect the variable from being overwritten by any accidental or stray assignments in your code.

```
const string firstName = "Jem";  
  
var age = 23;
```



```

Console.WriteLine(age);    //expected output: 23

age = 35;

Console.WriteLine(age);    //expected output: 35 as var let us reassign a value to
the age variable

firstName = "Jack";

Console.WriteLine(firstName); //expected output: error, as we cannot reassign a
constant variable value

```

Note: You cannot declare a constant variable without assigning the value and data type. If you do, an error will occur.

```

const firstName; //expected output: error

```

When declaring variables using the var keyword but omitting the data type, they also need to have a preliminary value -that is, to assign a value to it immediately. This is because C# will look at the type of the value the variable is initialized with, and use that as the type of the variable - this is known as being 'implicitly-typed'.

```

var age = 23;

Console.WriteLine(age); //expected output: 23 as age has implicitly typed type of int

```

If you declare a variable with the var keyword and data type, you can omit the value, and assign it later.

```

var int age;

```

### Type Assignment Method

The other method, which we will use going forward, includes specifying the data type, a name of the variable and then assigning it a value with the assignment operator.

```

string firstName = "Jem";

int age = 23;

```

You don't use the declaration keywords var or const with this method, however you should note that omitting these keywords gives the variable the default of var meaning the variable can be overridden.

### Naming A Variable



All C# variables must be identified with unique names ( called identifiers). A C# variable name must start with a letter, or underscore (\_) and then can be followed by any sequence of letters or numbers, but must never start with a number or contain spaces or special characters.

C#, like other programming languages, has reserved keywords that cannot be used as variable names such as class, static and more. To view all current, and potential future reservations, view Microsoft's reference.

It is best practice to give variables descriptive names, so that when someone looks back over the code it is easier to understand what it does. Because of this you may use variable names that contain multiple words. For this, you can use the aforementioned underscore or use camel case. Camel case is the act of capitalizing every word after the first word, like pricePerUnit.

### Variable Scope

A variable's scope is the visibility of the variable. This means what classes, methods, ect can see and therefore use the variable. For example, variables declared within a method are only accessible within that method. They are local variables. This allows you to use the same variable name in multiple methods without negatively affecting the outcomes of particular methods. Global variables are those accessible by multiple methods,ect, as they exist outside of a particular method scope.

### *Display Variables*

The most common way to display variables is with the WriteLine() method, but you can also use the Write() method. The difference between them is that WriteLine() prints the output on a new line each time, whilst Write() prints on the same line. With Write() you should add spaces where needed when outputting multiple variables at once, or concatenating them, as omitting spaces will place the variables directly next to each other.

```
Console.WriteLine("Hello!");

Console.WriteLine("Welcome to C#");

//expected output: Hello!

//Welcome to C#


Console.Write("Hello!");

Console.Write("Welcome to C#");

//expected output: Hello!Welcome to C#
```





## Data Types & Conversion

A data type specifies the type of values a variable can accept or are holding, and will be used by C# to determine what operations a variable can do.

```
string firstName = "Jem";           //string
char alphabet = 'a';                //character
int age = 23;                        //integer
long years = 7000000L;               //big integer
float BMI = 81.7F;                   //short decimal
double weight = 15.66;               //long decimal
bool onePlusOneEqualsTwo = true;     //boolean
```

| Data Type | Meaning  |
|-----------|--|
| string    | <b>String</b><br>Stores a sequence of characters, must be surrounded by double quotes          |
| char      | <b>Character</b><br>Stores a single character, must be surrounded by single quotes             |
| byte      | <b>Byte</b><br>A single byte in binary, represented in decimal from 0 to 255                   |
| short     | <b>Short Integer</b><br>Stores whole numbers. Accepts values from -32,768 to 32,768            |
| int       | <b>Integer</b><br>Stores whole numbers. Accepts values from -2,147,483,648 to 2,147,483,647    |
| long      | <b>Long Integer</b><br>Stores whole numbers. Accepts values from -9,223,372,036,854,775,808 to |



|        |  |
|--------|--|
|        | 9,223,372,036,854,775,807. Note that you should end the value with an "L"  |
| float  | <b>Short Decimal</b><br>Stores decimal (fractional) numbers. Can be used for storing 6 to 7 decimal digits. Note that you should end the value with an "F" |
| double | <b>Long Decimal</b><br>Stores decimal (fractional) numbers. Can be used for storing up to 15 decimal digits  |
| bool   | <b>Boolean</b><br>Stores true or false values  |

Types help prevent errors in code, such as trying to do a math expression with a string, this is known as type safety. This assures that when you have a value of a particular type, you can do particular operations on it, and be certain those operations will work.

### Type Casting

Type casting is when you assign a value of one data type to another type. There are two types of casting in C#, these are implicit and explicit.

#### *Implicit*

Implicit casting consists of automatically converting a smaller type to a larger type. It is done when passing a smaller size type to a larger size type.

Such as char to int, int to long or float to double.

```
int age = 23;

long years = age;           //automatic casting of int to long

Console.WriteLine(age);     //expected output: 23

Console.WriteLine(years);   //expected output: 23
```

#### *Explicit*

Explicit casting consists of manually converting a larger type to a smaller size type.

For example, double to float, or int to char.

It must be done manually by placing the type in parentheses in front of the value.



```
double decimalNum = 23.78;

int wholeNum = (int) decimalNum;    // manual casting of double to int

Console.WriteLine(decimalNum);    //expected output: 23.78

Console.WriteLine(wholeNum);        //expected output: 23
```

In C# there are also built-in methods for type conversion: `Convert.ToBoolean`, `Convert.ToDouble`, `Convert.ToString`, `Convert.ToInt32 (int)` and `Convert.ToInt64 (long)`.

```
int wholeNum = 23;

int truevar = 1;

int falsevar = 0;

double decimalNum = 23.78;

Console.WriteLine(Convert.ToBoolean(truevar));    //expected output: true
Console.WriteLine(Convert.ToBoolean(falsevar));    //expected output: false
Console.WriteLine(Convert.ToDouble(wholeNum));    //expected output: 23
Console.WriteLine(Convert.ToString(wholeNum));    //expected output: "23"
Console.WriteLine(Convert.ToInt32(decimalNum));    //expected output: 24
Console.WriteLine(Convert.ToInt64(decimalNum));    //expected output: 24
```

Data rarely has to be type converted, but in the instance of user input, the above comes in very useful.

### *Take Input*

Whilst `Console.WriteLine` is used to output values, `Console.ReadLine` is used to get user input.

In the below example we output the text 'What is your name?', and the input the user types is stored in a variable called `name`, meaning we can use this later in some other code.

```
Console.WriteLine("What is your name?");

string name = Console.ReadLine();
```

Alternatively, you can use `Console.ReadKey()`, this collects a single keystroke from the user.



The Console.ReadLine method returns a string, so if you wanted to collect user input of another type, say int age, you'd have to use one of the conversion methods above to collect the data in the correct type to be able to use it later.

```
Console.WriteLine("What is your age?");

int age = Convert.ToInt32(Console.ReadLine());

if (age < 18){

    Console.WriteLine("You're too young for this site!");

} else

{

    Console.WriteLine("Welcome!");

}
```

The above example asks the user for their age, and assigns their input to the age variable. If the user's age is less than 18, the 'too young' message is written to the console, else the 'welcome' message is shown.

## Operators

### Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on variables and values. If many arithmetic operators are used at once, BIDMAS will be used to decide the order of operation. Brackets, Indices, Division, Multiplication, Addition and Subtraction.

| Symbol   | Name & Meaning   | Example  |
|----------|--|--|
| <b>+</b> | <b>Addition</b><br>Adds numbers or variables containing numbers together         | <pre>5 + 5      //10 int x = 10, y = 20; x + y      //30</pre> |
| <b>-</b> | <b>Subtraction</b><br>Subtracts numbers or variables containing numbers together | <pre>5 - 5      //0 int x = 10, y = 20; x - y      //-10</pre> |



|           |   |   |
|-----------|---|---|
| <b>*</b>  | <b>Multiplication</b><br>Multiplies numbers or variables containing numbers together        | <pre>5 * 5 //25 int x = 10, y = 20; x * y //200</pre> |
| <b>/</b>  | <b>Division</b><br>Divides numbers or variables containing numbers                          | <pre>10 / 5 //2 int x = 36, y = 6; x / y //6</pre>    |
| <b>%</b>  | <b>Modulus</b><br>Returns the division remainder of numbers or variables containing numbers | <pre>10 % 5 //0 int x = 20, y = 6; x % y //2</pre>    |
| <b>++</b> | <b>Increment</b><br>Increases the value of a variable by 1                                  | <pre>int x = 20; x++; //21</pre>                      |
| <b>--</b> | <b>Decrement</b><br>Decreases the value of a variable by 1                                  | <pre>int x = 20; x--; //19</pre>                      |

### Assignment Operators

Assignment operators are used to assign values to variables.

| Symbol    | Example                                      |
|-----------|--|
| <b>=</b>  | <pre>int x = 10; //10 int y = 20; //20</pre> |
| <b>+=</b> | <pre>int x = 10; x += 5; //15</pre>          |
| <b>-=</b> | <pre>int x = 10; x -= 5; //5</pre>           |
| <b>*=</b> | <pre>int x = 10; x *= 5; //50</pre>          |
| <b>/=</b> | <pre>int x = 10; x /= 5; //2</pre>           |



**%=**

```
int x = 10;  
x %= 5; //0
```

Below are some more advanced assignment operators, to understand them you first must understand bitwise.

### Bitwise & Bitwise Operators

Bitwise is a level of operation that involves working with individual bits (the smallest units of data in a computer). Each bit has a binary value of 0 or 1. Bits are usually executed in bit multiples called bytes, with most programming languages manipulating groups of 8, 16 or 32 bits.

Bits can be complex for humans to understand, but are far easier to computers, as each bit (aka binary digit) can be represented by an electrical signal which is either on (1) or off (0).

See the below binary conversion table. The binary example below (11010) is equal to 26 in the decimal system (16+8+2=26).

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|-----|----|----|----|---|---|---|---|
|     |    |    | 1  | 1 | 0 | 1 | 0 |

Here is another example:

00001010 is the equivalent of 10 in the decimal system.

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|-----|----|----|----|---|---|---|---|
| 0   | 0  | 0  | 0  | 1 | 0 | 1 | 0 |

11110 would be the equivalent of 30.

Note: With binary notation you can perform some math operations very quickly:

- To half any number - simply move the digits 1 place to the right. Ei 11110 (30) -> 1111 (15)
- To double a number - simply add a zero on the end (shifts digits to the left). Ei 11010(26) -> 110100 (52)

Bitwise operators are useful to perform bit by bit operations.



| Symbol          | Name & Meaning   | Example   |
|-----------------|--|---|
| <b>&amp;</b>    | <b>Bitwise AND</b><br>This compares each individual bit of the first operand with the corresponding bit of the second operand. If both bits are 1, then the result bit will be 1 otherwise it will be 0  | <pre>int a = 10;           //(00001010) int b = 20;           //(00010100) Console.WriteLine(a &amp; b); //(00000000) equal to 0 Console.WriteLine(a);   //(00001010) equal to 10</pre> |
| <b>&amp;=</b>   | The &= operator compares the bits of the operand on its right to the operand on its left, and assigns the result to the operand on its left  | <pre>int a = 10; //(00001010) int b = 20; //(00010100) a &amp;= b; //assigns result to a Console.WriteLine(a); //(00000000) equal to 0</pre>  |
| <b> </b>        | <b>Bitwise OR</b><br>This compares each individual bit of the first operand with the corresponding bit of the second operand. If either of the bits is 1, then the result bit will be 1 otherwise the result will be 0                               | <pre>int a = 10;           //(00001010) int b = 20;           //(00010100) Console.WriteLine(a   b); //(00011110) equal to 30 Console.WriteLine(a);   //(00001010) equal to 10</pre>    |
| <b> =</b>       | The  = operator compares the bits of the operand on its right to the operand on its left, and assigns the result to the operand on its left  | <pre>int a = 10; //(00001010) int b = 20; //(00010100) a  = b; Console.WriteLine(a); //(00011110) equal to 30</pre>   |
| <b>^</b>        | <b>Bitwise Exclusive OR (XOR)</b><br>It compares each individual bit of the first operand with the corresponding bit of the second operand. If one of the bits is 0 and the other is 1, then the result bit will be 1 otherwise the result will be 0 | <pre>int a = 10;           //(00001010) int b = 20;           //(00010100) Console.WriteLine(a ^ b); //(00011110) equal to 30 Console.WriteLine(a);   //(00001010) equal to 10</pre>    |
| <b>^=</b>       | The ^= operator compares the bits of the operand on its right to the operand on its left, and assigns the result to the operand on its left  | <pre>int a = 10; //(00001010) int b = 20; //(00010100) a ^= b; Console.WriteLine(a); //(00011110) equal to 30</pre>   |
| <b>&lt;&lt;</b> | <b>Bitwise Shift Left</b><br>This shifts bits to the left by the amount specified on the right. E.g. shift of 1, moves the bits left 1 position each (end will have a zero)  | <pre>int b = 20;           //(00010100) Console.WriteLine(b &lt;&lt; 2); //(1010000) equal to 80 // (2 is equivalent of multiplying by 4)</pre>   |



|     |  |   |
|-----|--|---|
|     | added to it). This is equivalent to multiplication by 2  | <pre>int b = 20;           //(00010100) Console.WriteLine(b &lt;&lt; 1 ); //(101000) equal to 40</pre>    |
| <<= | The <<= operator shifts the bits of the operand on its left by the amount specified on the right, and assigns the result to the operand on its left  | <pre>int b = 20;           //(00010100) b &lt;&lt;= 1; Console.WriteLine(b); //(101000) equal to 40</pre> |
| >>  | <b>Bitwise Shift Right</b><br>This shifts bits to the right by the amount specified on the right. Ei shift of 1, moves the bits right 1 position each. This is equivalent to division by 2 | <pre>int b = 20;           //(00010100) Console.WriteLine(b &gt;&gt; 1 ); //(00001010) equal to 10</pre>  |
| >>= | The >>= operator shifts the bits of the operand on its left by the amount specified on the right, and assigns the result to the operand on its left  | <pre>int b = 20; //(00010100) b &gt;&gt;= 1; Console.WriteLine(b); //(00001010) equal to 10</pre>         |

The below operand is used to flip bits, ei 0 becomes 1 and vis versa.

| Symbol | Name & Meaning   | Example  |
|--------|--|--|
| ~      | <b>Bitwise Complement</b><br>This operates on only one operand, and it will invert each bit of the operand (ei it will change 1 to 0 and 0 to 1) | <pre>int b = 20; //(00010100) b =~(b); Console.WriteLine(b); // -21 (11101011)</pre> |

## Selection

With conditional statements you'll often use logical and or comparison operators to test a condition to be able to output(run) particular code blocks if the condition is true, and another if it is false.

| Symbol | Name & Meaning  | Example   |
|--------|---|---|
| ==     | <b>Equal To</b><br>Tests whether expression on it's left is equal to that on it's right | <pre>bool z = 5 == 6; Console.WriteLine(z);</pre> |





|                   |  |   |
|-------------------|--|---|
|                   |  | <pre>//expected output: false</pre>   |
| <b>!=</b>         | <b>Not Equal To</b><br>Tests whether expression on it's left is not equal to that on it's right  | <pre>bool z = 5 != 6; Console.WriteLine(z); //expected output: true</pre>                         |
| <b>&gt;</b>       | <b>Greater Than</b><br>Tests whether expression on it's left is greater than that on it's right  | <pre>bool z = 5 &gt; 6; Console.WriteLine(z); //expected output: false</pre>                      |
| <b>&lt;</b>       | <b>Less Than</b><br>Tests whether the expression on it's left is less than that on it's right. A way to remember the difference between greater than and less than is that less thans symbol looks like an 'L' (for less than) | <pre>bool z = 5 &lt; 6; Console.WriteLine(z); //expected output: true</pre>                       |
| <b>&gt;=</b>      | <b>Greater Than Or Equal To</b><br>Tests whether expression on it's left is greater than, or equal to that on it's right   | <pre>bool z = 5 &gt;= 6; Console.WriteLine(z); //expected output: false</pre>                     |
| <b>&lt;=</b>      | <b>Less Than Or Equal To</b><br>Tests whether expression on it's left is less than, or equal to that on it's right   | <pre>bool z = 5 &lt;= 6; Console.WriteLine(z); //expected output: true</pre>                      |
| <b>&amp;&amp;</b> | <b>Logical AND</b><br>Returns true if both statements are true, else returns false   | <pre>bool z = (5 == 5) &amp;&amp; (5 &lt; 6); Console.WriteLine(z); //expected output: true</pre> |
| <b>  </b>         | <b>Logical OR</b><br>Returns true if one of the statements are true, else returns false  | <pre>bool z = (5 == 5)    (5 &lt; 6); Console.WriteLine(z); //expected output: true</pre>         |
| <b>!</b>          | <b>Logical NOT</b><br>Reverses the result - it returns false if the result is true, and true if the result is false  | <pre>bool z = !(5 == 5); Console.WriteLine(z); //expected output: false</pre>                     |



## The If Statement

The if statement is used to specify a block of code to be executed if the condition within it's parentheses () is true.

```
if (condition){  
    code to execute if condition is true  
}
```

The below example asks the user for their age, and assigns their input to the age variable. If the user's age is less than 18, the 'too young' message is written to the console.

```
Console.WriteLine("What is your age?");  
  
int age = Convert.ToInt32(Console.ReadLine());  
  
if (age < 18){  
    Console.WriteLine("You're too young for this site!");  
}
```

We can then use the else statement to specify a code block to execute if the condition is false.

```
if (condition){  
    code to execute if condition is true  
} else  
{  
    code to execute if condition is false  
}
```

```
Console.WriteLine("What is your age?");  
  
int age = Convert.ToInt32(Console.ReadLine());  
  
if (age < 18){  
    Console.WriteLine("You're too young for this site!");  
}
```



```

} else

{

    Console.WriteLine("Welcome!");

}

```

The above example asks the user for their age, and assigns their input to the age variable. If the user's age is less than 18, the 'too young' message is written to the console, else the 'welcome' message is shown.

If we wanted to test another condition if the first evaluates to false, we can add an else if statement after the initial if statement, with a new condition to test in it's parentheses.

```

if (condition){

    code to execute if condition is true

} else if (second_condition){

    code to execute if condition is true

} else {

    code to execute if condition and second_condition are false

}

```

You can add as many else if statements as you want, however for a conditional statement that has multiple conditions to test, it is best to use the switch statement instead.

### The Ternary Operator

The ternary operator is used as a shorthand for if else statements. It is known as the ternary operator because it consists of three operands- see the syntax below.

```

variable = (condition) ? outputForTrue_Expression_Evaluation :
outputForFalse_Expression_Evaluation;

```

The above syntax assigns the ternary expression outcome to a variable, but you can just as easily use it with methods by replacing the 'variable =' part with the return keyword.

The below is the previous if else example reconstructed in the ternary operator format.

```

int age = Convert.ToInt32(Console.ReadLine());

```



```
string greeting = (age < 18) ? "You're too young for this site!" : "Welcome";  
Console.WriteLine(greeting);
```

### *The Switch Statement*

The switch statement is used to select one of many code blocks to be executed based on cases.

```
switch(expression)  
{  
    case A:  
        code to execute if case A is true  
        break;  
    case B:  
        code to execute if case B is true  
        break;  
    default:  
        code to execute if NON of the cases above are true  
        break;  
}
```

With the switch statement, the expression is evaluated once and the value of the expression is compared with the values of each case. If a case value matches, then it's associated code block will be executed.

The break keyword is important because when C# reaches a break keyword, it breaks out of the switch block, which stops the execution of more code and case testing inside the switch block, and stops unexpected behaviour from happening.

The default keyword is optional. It is used to specify some code to run if there is no case match.

The below example of a switch statement first asks the user for their favourite animal, to which their input is stored in a variable called 'favAnimal'. As you can see, their input is also converted



to all lowercase letters, so that even if they type 'DOG' or 'DoG', they would still get the correct switch case of 'dog'.

The switch evaluates the users input from the 'favAnimal' variable and compares it to it's available cases. Ei if the user input 'cat' then this matches the 'cat' case, and it's code block is executed.

You may also notice that there is one code block for multiple 'bear' type cases. This is fine. If you want one particular code block to run for more than one case, instead of copy and pasting the code block over and over again, you can just place the cases one after another, with the last case holding the code to execute if any of the cases defined match.

```
Console.WriteLine("What\'s your favourite animal?");  
  
string favAnimal = Console.ReadLine().ToLower();  
  
switch (favAnimal){  
  
    case "dog":  
  
        Console.WriteLine("I love dogs too!");  
  
        break;  
  
    case "cat":  
  
        Console.WriteLine("I love cats too!");  
  
        break;  
  
    case "panda":  
  
    case "bear":  
  
    case "red panda":  
  
    case "polar bear":  
  
        Console.WriteLine("I love all species of bears!");  
  
        break;  
  
    default:  
  
        Console.WriteLine("Animals are great!");  
  
        break;
```



```
}
```

## Iteration

### *Loops*

Loops are used to repeatedly execute a block of code, so long as a condition is true. Once it becomes false the execution stops. When working with loops it's important to create a condition or expression that will eventually become false, otherwise the loop will never end. You can do this by incrementing or decrementing a variable until it reaches a point where the expression using it becomes false.

### While

The while loop loops through a code block as long as it's condition is true.

```
while (condition)
{
    code to execute if condition is true
}
```

The below example prints the numbers 0-4.

```
int i = 0;
while (i < 5) {
    Console.WriteLine(i);
    i++;}

/*expected output:
0
1
2
3
4 */
```

### Do While



The do while loop is a variant of the while loop. The difference between them is that the do while loop will execute the code block once before testing the condition. Then it will repeat the loop as long as the condition is true.

```
do {  
    code to execute  
}  
while (condition);
```

With do while, the loop will always be executed at least once- even if the condition is false- because the code block is executed before the condition is tested.

```
int i = 5;  
  
do{  
    Console.WriteLine(i);  
    i++;  
}  
while (i < 5);  
  
//expected output: 5
```

## For

The for loop is useful when you know exactly how many times you want to loop through a code block. It's syntax is as below.

```
for (statement1; condition; statement3)  
{  
    code to execute  
}
```

**statement1** is evaluated once before the execution of the code block.

**condition** is the condition to be evaluated, it defines the condition required to execute the code block.



**statement3** is evaluated every time, after the code block has been executed. This is the statement that is usually used to increment or decrement a variable to ensure that the condition will eventually evaluate to false.

```
for (int i = 1; i < 5; i++)
{
    Console.WriteLine(i);
}

/*expected output:
1
2
3
4 */
```

In the example above, a local variable called *i* is created and initialized with the value of 1. The value is incremented by 1 every time the loop runs, and prints the increasing variable each time to the console. The loop is set to run as long as *i* is less than 5, once it reaches 5 the loop stops.

## Foreach

Foreach is used exclusively to loop through elements in an array.

```
foreach (type variableName in arrayName)
{
    code to execute
}
```

**variableName** here is a local variable, meaning that you can name it anything you want - as long as it follows standard variable naming convention.

The below example prints all elements in the *animals* array to the console. The *foreach* statement below can be read as 'for each string element called *i* (local variable) in the *animals* array, print the value of *i*.'

```
string[] animals = {"bear", "bat", "bee", "beetle"};
```





```
foreach ( string i in animals) {
    Console.WriteLine(i); }

/*expected output:

bear

bat

bee

beetle

*/
```

## Break & Continue

As well as being used in the switch statement to break out of the switch, break can also be used to jump out of loops.

```
for (int i = 1; i < 8; i++) {
    if (i == 4) {
        break;
    }

    Console.WriteLine(i);
}

/*expected output:

1

2

3

*/
```

The above example breaks out of the loop once the value of the local variable i reaches 4.

The continue keyword on the other hand, is used to break one iteration in the loop, if a specified condition occurs. It then continues with the next iteration in the loop.



```

for (int i = 1; i < 8; i++) {
    if (i == 4) {
        continue;
    }

    Console.WriteLine(i);
}

/*expected output:
1
2
3
5
6
7
*/

```

The above example, 'skips' over printing the value of 4, and continues on with the loop until it's condition becomes false.

### *Recursion*

Recursion is a technique that causes a method to call itself in order to compute a result. Similar to loops, a recursive method must terminate eventually.

```

using System;

public class Program
{
    public static int RecursiveSumExample(int n, int m)
    {
        int sum = n;
    }
}

```



```

        if(n < m) //if n is less than m

        {

            n++;

            return sum += RecursiveSumExample(n, m); //same as sum = sum + n

        }

        return sum;

    }

    public static void Main()

    {

        Console.WriteLine("Enter number n: ");

        int n = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("Enter number m: ");

        int m = Convert.ToInt32(Console.ReadLine());

        int Sum = RecursiveSumExample(n, m);

        Console.WriteLine(Sum);

    }

}

```

Take the above example, and say the input for n is 1 and m is 3. The below will show how the recursion loop works. M is used as the finitor, and when n is equal to m, the recursion loop stops.

Sum is calculated 3 times in this example, each time increasing in value as it has the assigned value of n. So when n = 1, sum is also 1.



| When n = | sum = |
|----------|-------|
| 1        | 1     |
| 2        | 2     |
| 3        | 3     |

The loop is now complete. As n is no longer less than m.

Each recursion now backtracks ( `return sum += RecursiveSumExample(n, m);` //same as `sum = sum + n` ) . sum is returned with each recursion added (1+2+3) to give sum the final return value of 6.



When  $n = 1$  &  $m = 3$

```
public static int RecursiveSumExample(int n, int m)
{
    int sum = n; sum = 1
    if(n < m) //if n is less than m true
    {
        n++; Now n = 2
        return sum += RecursiveSumExample(n, m); //same as sum = sum + sum
    }
    return sum; sum = 3+2+1
}
```

```
public static int RecursiveSumExample(int n, int m) repeat
{
    int sum = n; sum = 2
    if(n < m) //if n is less than m true
    {
        n++; Now n = 3
        return sum += RecursiveSumExample(n, m); //same as sum = sum + sum
    }
    return sum; sum = 3+2
}
```

```
public static int RecursiveSumExample(int n, int m) repeat
{
    int sum = n; sum = 3
    if(n < m) //if n is less than m false
    {
        n++;
        return sum += RecursiveSumExample(n, m); //same as sum = sum + sum
    }
    return sum; sum = 3
}
```

Some more examples using the above method:

When  $n =$

When  $m =$

Final sum =



|   |    |   |
|---|----|---|
| 1 | 5  | 15<br>(5 + 4 + 3 + 2 + 1)                         |
| 1 | 10 | 55<br>(10 + 9 + 8 + 7 + 6 + 5 + 4<br>+ 3 + 2 + 1) |
| 5 | 10 | 45<br>(10 + 9 + 8 + 7 + 6 + 5)                    |
| 7 | 10 | 34<br>(10 + 9 + 8 + 7)                            |

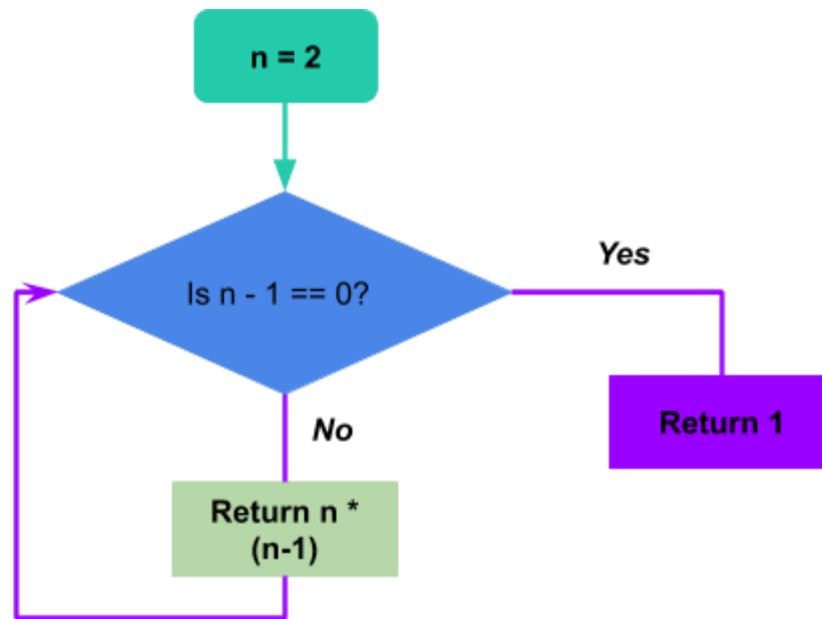
Another example using the new method below.

```
public class Program
{
    public static int Factorial(int n)
    {
        if(n == 0)
        {
            return 1;
        } else {
            return n * Factorial(n-1);
        }
    }
}

public static void Main()
{
    Console.WriteLine("Enter number n: ");
    int n = Convert.ToInt32(Console.ReadLine());
    int sum = Factorial(n);
    Console.WriteLine(sum);
}
}
```

In factorials, the number value is multiple by its previous number. Say for example n was 2:





So if  $n = 2$ . The result will be  $2 (2 \times 1 \times 1)$ .

This is because  $n$  is decreased each iteration.  $2-1 = 1$ , then  $1-1$  is equal to 0 so we return 1.

Again, here is more examples using the above method:

| When $n =$ | sum =   |
|------------|---|
| 5          | 120<br>( $5 \times 4 \times 3 \times 2 \times 1 \times 1$ )   |
| 8          | 40320<br>( $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1$ )                      |
| 10         | 3628800<br>( $10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1$ ) |

## Methods

A method is a group of code statements that are put together to perform a particular task (they are also called functions).

### Create A Method



To create a method define the name of the method, followed by parentheses () and then a pair of curly braces. These curly braces are the method body and hold one+ lines of code that will be run when the method is called.

Method names follow the same naming conventions as naming variables, however programmers typically start with an uppercase letter for method names. A method signature is it's name and parameters.

```
static void AMethod()  
{  
  
}
```

Method names will be prepended with keywords such static and return values such as void.

The **static** keyword means you can call the method by itself - it doesn't belong to an object, GetType() is an example of a method you can only call on an object, where you must place a dot operator after an object, then call the method.

### Static VS Instance Methods

An example of a static method is the WriteLine() method. Console is the name of the class, and we call the WriteLine() method directly on the Console class without constructing a console object.

```
Console.WriteLine(FRUIT);
```

Compare this to the instance method below: The sound method is not static. And so we had to construct an animal class object (called dog), to then be able to call sound().

```
...  
public void sound(string sound)    // method  
{  
    Console.WriteLine($"The sound I make is: {sound}!");  
}}  
  
public class Program  
{  
    public static void Main(string[] args)
```





```

{
    animal dog = new animal();

    dog.sound("woof"); //The sound I make is: woof!
}

```

It's important to remember that methods that are called directly on a class name are static whilst instance methods are called on unique instances(objects) of the class.

Also note that, in the case of static classes only one copy of a static field is shared by all instances, because if a whole class is static, then it cannot be instantiated and therefore there is only ever one of that type.

### Return Values

void means that the method has no return value. If the method should return a value, use the data type of the value it should return instead of the void keyword. The return keyword should also be used within the method.

```

static int AMethod(int x, int y)
{
    return x*y;
}

public static void Main(string[] args)
{
    Console.WriteLine(AMethod(5,2));
}

//expected output: 10

```

The example above returns an int value.

### Call A Method

To call (run) a method, write the method's name followed by two parentheses () and a semicolon.

```
AMethod(5,2);
```



## Parameters & Methods

Information can be passed to methods as parameters - which act as variables inside the method. Parameters are declared inside the method parentheses, and follow variable naming convention.

You can add as many parameters as you want, but they must be separated by a comma.

```
static void AMethod(string name, int age)
{
    Console.WriteLine($"Hello {name}! You are {age} years old." );
}

public static void Main(string[] args)
{
    AMethod("Jem",23);
}

//expected output: Hello Jem! You are 23 years old.
```

Note: you may hear people use parameter and argument interchangeably, this is because when a parameter value is passed to the method, this is called an argument. In the above example, name and age are parameters whilst 'Jem' and '23' are arguments.

Parameters can accept default values by using the assignment sign after the parameter.

For example, below gives name the default of 'friend'.

```
static void AMethod(string name = "friend", int age)
```

When using multiple parameters it is important to remember that the method call must have the same number of arguments as there are parameters. Arguments should also be passed in the same order as they are declared, however it is possible to pass a method arguments in key:value pairs which means the order doesn't matter.

```
AMethod(age:23,name:"Jem");

//expected output: Hello Jem! You are 23 years old.
```

Named arguments like above are useful when you have multiple parameters with default values, and you only want to pass one, or a few, none default values when you call the method.



## Collections

Collections in C# are used to organize data so that it can be used efficiently.

### *Arrays & Multi Dimensional Arrays*

Arrays are used to store multiple values in a single variable.

#### Create An Array

There are many ways to create an array, one way is to declare the variable type, followed by a pair of square brackets and then the array name.

```
string[] animals; //creates a string array called animals  
int[] years;      //creates a int array called years
```

Values can then be added to this array declaration type later by creating an array literal (a comma separated list of values inside a pair of curly braces).

```
string[] animals = {"bear", "bat", "bee", "beetle"};
```

Other ways to create arrays include:

```
// Creates a string array called animals, which can hold four elements. These  
elements can be added later  
string[] animals = new string[4];  
  
// Creates a string array of four elements and initializes it with the values  
string[] animals = new string[4] {"bear", "bat", "bee", "beetle"};  
  
// Creates an array, initialized with four elements, but without specifying the  
size  
string[] animals = new string[] {"bear", "bat", "bee", "beetle"};
```

#### Access Elements In An Array

We can access elements in an array using the index of the element. This includes writing the array name, followed by a pair of square brackets, and in said brackets writing the index number of the element you want to retrieve - starting at 0 for the first element in the array, 1



for the second, and so forth. If you try to access an index that does not exist in the array, you'll get a 'index out of range' exception.

```
string[] animals = {"bear", "bat", "bee", "beetle"};

Console.WriteLine(animals[0]);

//expected output: bear

Console.WriteLine(animals[2]);

//expected output: bee
```

### Change Array Elements

We can also change elements in an array using indexes.

```
string[] animals = {"bear", "bat", "bee", "beetle"};

animals[0] = "dog";

foreach ( string i in animals) {

    Console.WriteLine(i); }

/*expected output:

dog

bat

bee

beetle

*/
```

### Find Array Length

We can find an array's length using the .Length property. This property can also be used to select the last element in an array - demonstrated below.

```
Console.WriteLine(animals[animals.Length-1]); //expected output: beetle
```

### Looping Through Arrays

You can loop through array elements with the for loop, by using the .Length property to specify how many times the loop should run. Demonstrated below.



```

string[] animals = {"bear", "bat", "bee", "beetle"};

for (int i = 0; i < animals.Length; i++)
{
    Console.WriteLine(animals[i]);
}

/*expected output:

bear

bat

bee

beetle

*/

```

However it is better to use the [foreach loop](#) for looping through arrays.

### Multidimensional Arrays

Two dimensional arrays are similar to tables, with rows and columns.

They are defined similarly to standard arrays, however, in the square brackets we add a comma(,) to separate the dimensions.

The below example creates an int array with 10 rows and 4 columns.

```
int[,] multiExample = new int[10,4];
```

The above array holds 40 items.

```
Console.WriteLine(multiExample.Length); //40
```

We access multidimensional arrays in a similar format to standard.

```

int[,] multiExample = new int[10,4];

multiExample[1,1] = 23;

Console.WriteLine(multiExample[1,1]); //23

Console.WriteLine(multiExample[1,2]); //0

```



```
string[,] twoDimensions = new string[3,2];

twoDimensions[1,1]= "Jem";

Console.WriteLine(twoDimensions[1,1]); //Jem

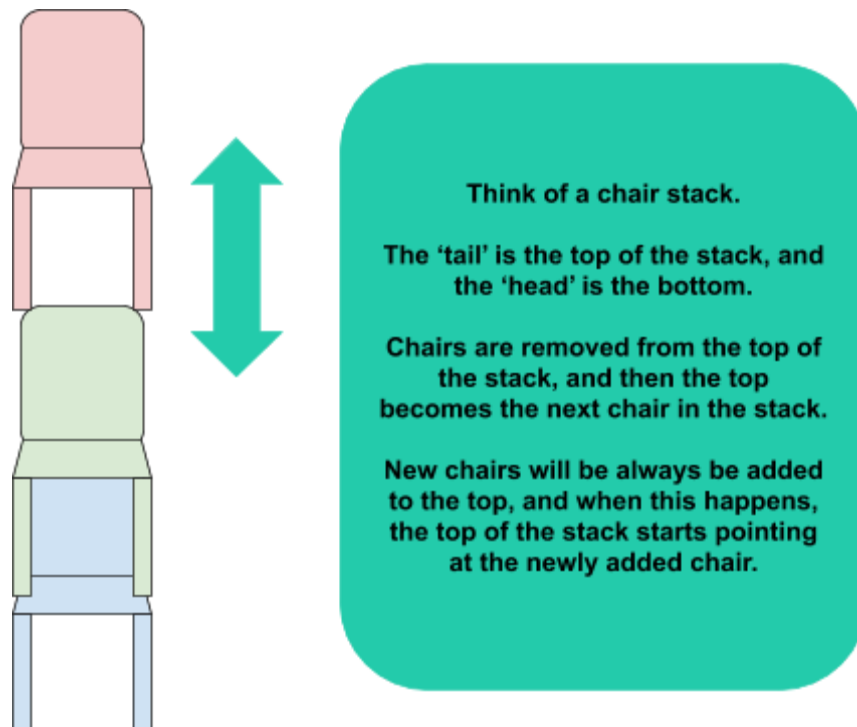
Console.WriteLine(twoDimensions[1,0]); //null
```

### Stacks

A stack is a collection of items, where the last item added to the collection is the first one to be removed. This is referred to as 'Last In First Out (LIFO)'.

A stack is a heterogeneous data structure, meaning that it is able to contain a variety of data, including those of different data types. For example, a stack can hold a mix of integer, float and character data. This is the opposite of an array. Which is an homogeneous data structure, meaning that it can contain data of only a similar type. For example, like above created an array of integers - it can only hold integers.

The capacity of a stack is also different to that of an array, in that it is automatically increased as we add more items.



## Create A Stack

To create a stack, add the using System.Collections namespace, then you can create a new stack but using the Stack keyword, followed by a name for the stack, the assignment operator, the new keyword and then Stack().

To add items to a stack use the push() method. This adds an item to the top of the stack.

```
using System;

using System.Collections;

public class Program
{
    public static void Main()
    {
        Stack myExample = new Stack();

        myExample.Push("Jem");

        myExample.Push(23);

    }
}
```

## Access Elements In A Stack

You can access elements in a stack using the peek() method. This allows you to access the current item at the top of the stack, without removing it directly from the stack.

```
Console.WriteLine(myExample.Peek()); //23
```

You can also use a foreach loop, to retrieve the items in a stack.

```
public static void Main()
{
    Stack myExample = new Stack();
```



```

myExample.Push("Jem");

myExample.Push(23);

foreach (var stackItem in myExample){

    Console.WriteLine(stackItem);

}

}

//23

//Jem

```

Notice how it starts from the top of the stack, and prints 23 before Jem? Also note how we specified 'var' instead of a specific data type in our loop, this is because our stack holds different data types, and so our foreach loop needs to be implicitly typed.

We can also use the pop() method to retrieve an item. This method returns (and removes) the top element of the stack.

The contains() method checks whether the specified items exists in a stack. It returns a boolean value of true if the item exists, and false if not.

```

Console.WriteLine(myExample.Contains("Lorenzo")); //false

Console.WriteLine(myExample.Contains(23)); //true

```

The clear() method is used to remove all elements in a stack. You can then check this is true by using the count property of stacks, which returns the total count of elements in a stack.

```

myExample.Clear();

Console.WriteLine(myExample.Count); //0

```

## Queues

A queue is a collection of items where the first item added to the collection is the first to be removed. This is referred to as 'First In First Out' (FIFO).

Like stacks, queues are heterogeneous data structures. And the capacity of the queue (the number of items it can hold), is automatically increased as items are added.

### Create A Queue





The process to create a queue is similar to that of a stack, see below.

To add elements to a queue use the enqueue() method. This adds elements to the end of the queue.

```
using System;

using System.Collections;

public class Program
{
    public static void Main()
    {
        Queue myExample = new Queue();

        myExample.Enqueue("Jem");

        myExample.Enqueue(23);

    }
}
```

### Access Elements In A Queue

The dequeue() method is used to return and remove the first element from a queue.

Whilst peek() returns the first element in a queue without removing it.

We can also iterate through elements in a queue using the foreach method, or check whether the queue contains an item using the contains() method.

```
Queue myExample = new Queue();

myExample.Enqueue("Jem");

myExample.Enqueue(23);

    foreach (var stackItem in myExample){

        Console.WriteLine(stackItem);

    }
```



```

    }

    //Jem

    //23

Console.WriteLine(myExample.Peek()); //Jem

Console.WriteLine(myExample.Contains("Lorenzo")); //false

Console.WriteLine(myExample.Contains(23)); //true

```

The `clear()` method and `count` property can also be used with a queue.

```

myExample.Clear();

Console.WriteLine(myExample.Count); //0

```

### *Lists*

A list in C# is like an array, however it is more flexible, as it dynamically resizes based on the element count within it.

The capacity of a list increases by four each time we reach the threshold. For example, if we had 3 items in a list (resulting in our current capacity being 4), but then added two more, our stack would increase its capacity to 8. This is because the list collection is actually a wrapper around an array. All our 'list' items are stored in an array, and when we first declare the list, the array is null. When the first item is added, an array length of 4 is created to store that item and upto 3 others. When we added a 5th item there wasn't enough capacity to store it, so instead, List created a new array with a length of 8 and copied all the items from the original array to the larger array. If we were using an array from the start, we'd have to manually increase the size if we wanted more capacity, but with Lists the capacity increases automatically.

### Create A List

To create a list, use the `List` keyword followed directly by a pair of angle brackets (`<>`), within these brackets place the data type of the elements the list will hold, then follow this with the name for the list, an assignment operator, the `new` keyword and `List<data_type>()`.

The List collection exists in the `System.Collections.Generic` namespace.

```

using System;

using System.Collections.Generic;

```



```
public class Program
{
    public static void Main()
    {
        List<string> myExample = new List<string>();
        myExample.Add("Jem");
    }
}
```

To add items to a list, use the add() method.

```
myExample.Add("Jem");

myExample.Add(23); //error

myExample.Add("Jack");
```

Lists are homogeneous, notice we cannot add an integer to our string list.

Alternatively use the initialiser syntax, similar to that of an array.

```
List<string> myExample = new List<string>() {"Ben", "Beu"};
```

In a list, items are in the order they are added.

### Access Elements In A List

We can access elements in a list using the foreach loop. Or by using the index method.

```
List<string> myExample = new List<string>() {"Ben", "Beu"};

myExample.Add("Jem");

myExample.Add("Jack");

    foreach (var stackItem in myExample){

        Console.WriteLine(stackItem);

    }
```



```
/*Ben  
Beu  
Jem  
Jack  
*/  
  
Console.WriteLine(myExample[2]); //Jem
```

Alternatively we can use a for loop to iterate over the elements by combining it with the .Count property.

We can insert elements into a list at a specific index using the insert() method.

```
myExample.Insert(3, "Jessy");
```

The above inserts "Jessy" into index position 3 (this is the position after Jem).

We can remove items from a list using the remove() and removeat() methods.

With remove(), you specify the item you want to remove, whilst with removeat() you specify the index of the position you want to remove.

```
myExample.Remove("Jessy");  
  
myExample.RemoveAt(3);
```

Both the above remove "Jessy" from our list.

To check if a list contains an item, use the contains() method, and to check the capacity of a list use the capacity property.

```
Console.WriteLine(myExample.Contains("Lorenzo")); //false  
  
Console.WriteLine(myExample.Capacity); //8
```



## Objects & Classes

As previously mentioned, C# is an Object Oriented Programming Language.

Classes and objects are key aspects of any OOP.

A class is a template for objects, whilst an object is an instance of a class. When an object is created, it inherits all variables and methods of the class.

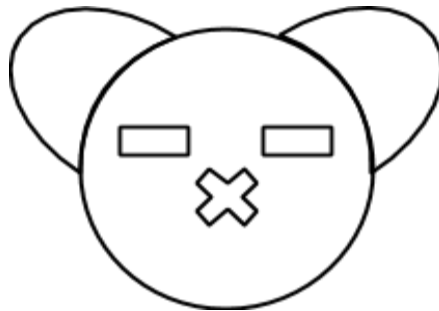
Objects are self contained data structures that consist of properties (which specify the data represented by an object) and methods (which specify an objects behaviour).

### *Create A Class*

In C# every value that a variable can take on is an object, for example 5 and 23 are objects of type int, and "Jem" and "bear" are objects of type string. These are types built into C#. But we can make our own types, with classes.

A class is a template for making individual objects of a particular type.

For example, think of a cookie cutter. The classes are the cookie cutter itself whilst objects are the cookies ( remember - objects are instances of a class).



This cookie cutter is bear shaped, and so every cookie it produces will be of that type. Each cookie created is a distinct (individual) object. Even though they will all be shaped like bears.

And just because we are using the same cutter, doesn't mean the objects will all be the same. We could use different ingredients or add coloring. In this instance color, and ingredients would be attributes of our cookie object.



We can create different objects from a class by changing things like the attributes of the class.

### Create A Class

To create a class, use the class keyword, followed by the name of the class and a pair of curly braces.

```
class animal
{
    int eyes = 2;
}
```

The above creates an 'animal' class, with an int variable inside of it called eyes. When a variable is declared directly in a class, like above, it is referred to as a field or attribute.

### *Create A Object*

Since an object is created out of a class, we can use the animal class created previously to create a new object.

To create an object of a class, specify the class name, followed by the object name, an assignment operator, and the new keyword, followed then by the class name again with parentheses.

```
class animal
{
    int eyes = 2;

    static void Main(string[] args)
    {
        animal myAnimal = new animal();

        Console.WriteLine(myAnimal.eyes); //2
    }
}
```



The above example creates one new animal object. We could create many by repeating the creation steps.

Fields and methods inside of classes are often referred to as class members. We can access fields through the dot(.) notation, demonstrated above. But we can also leave fields blank, and assign values to them when creating the objects.

```
class animal
{
    string color;

    int eyes;

    static void Main(string[] args)
    {
        animal bear = new animal();

        bear.color = "brown";

        bear.eyes = 2;

        Console.WriteLine(bear.color); //brown

        Console.WriteLine(bear.eyes);  //2

    }
}
```

Methods, which normally belong to classes and define how an object of a class behaves, can also be accessed by the dot(.) notation. However class methods must have the public access modifier instead of static to do so, as a static method can be accessed without creating an object of the class, while public methods can only be accessed by objects.

```
class animal
{
    string color;           // field

    int eyes;               // field

    public void sound(string sound) // method
}
```



```

    {
        Console.WriteLine($"The sound I make is: {sound}!");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        animal dog = new animal();

        dog.sound("woof"); //The sound I make is: woof!
    }
}

```

### *Multiple Classes*

We can also create an object of a class and access it in other classes. This practice is often used by programmers for better organization of classes, where one class will have all the fields and methods, whilst another class will hold the Main() method.

```

class animal
{
    public int eyes = 2;
}

public class Program
{
    public static void Main(string[] args)
    {
        animal myAnimal = new animal();

        Console.WriteLine(myAnimal.eyes);
    }
}

```





```

    }
}

```

The **public** keyword is vital here, as it is the access modifier that specifies that the eyes variable(field) of the animal class is accessible for other classes as well, such as program in this instance.

### Access Modifiers

An access modifier is used to set the access level (visibility) for classes, fields, methods and properties. The default modifier is private.

| Access Modifier | Meaning   |
|-----------------|---|
| public          | Code is accessible for all classes  |
| private         | Code is only accessible within the existing (same) class                                  |
| protected       | Code is accessible within the same class, or in a class that is inherited from that class |
| internal        | Code is only accessible within its own assembly   |

### Encapsulation

Encapsulation is the process of making sure that "sensitive" data is hidden from users. To encapsulate data we must declare fields(variables) as private, and provide public get and set methods through properties, to access and update the value of a private field.

Private variables can be accessed through properties, which are a combination of a variable and method.

See the below example, in which the Address **property** is associated with the private address **field**. Note: programmers typically use the same name for both the property and the private field, but with an uppercase first letter for the property.

The get method, here, returns the value of the variable address. Whilst the set method assigns a value to the address field.

```

class Person
{
    private string address;    // field

```



```

public string Address    // property
{
    get { return address; }    // get method
    set { address = value; }    // set method
}
}

```

Below is an example of encapsulation in action.

```

class Person
{
    private string address;    // field

    public string Address    // property
    {
        get { return address; }    // get method
        set { address = value; }    // set method
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Person jem = new Person();

        jem.Address = "23 bear street";
    }
}

```



```

        Console.WriteLine(Jem.Address); //23 bear street
    }
}

```

The encapsulation allows us to change the value of the protected variable whilst still securing the data.

```

Person Jem = new Person();

Console.WriteLine(Jem.Address); //blank - as no value assigned yet

Jem.Address = "23 bear street";

Console.WriteLine(Jem.Address); //23 bear street

Jem.Address = "43 cat grove";

Console.WriteLine(Jem.Address); //43 cat grove

```

Encapsulation also gives better control of class members by reducing the possibility of people messing up the code, by allowing us to make fields read-only - by only using the get method - or write-only - by only using the set method.

### *Properties Short-Hand*

C# also has a short-hand available for creating properties, in that you do not define the field for the property, instead you only have to write get; and set; inside the property.

The below example works the same as the previous.

```

class Person
{
    private string address;    // field

    public string Address      // property
    {
        get; set;
    }
}

```

```

}

public class Program
{
    public static void Main(string[] args)
    {
        Person Jem = new Person();

        Jem.Address = "23 bear street";

        Console.WriteLine(Jem.Address); //23 bear street
    }
}

```

With the shorthand syntax, this becomes a 'auto-implemented' property in C#, meaning that the .NET framework automatically creates a string variable behind the scenes for Address, for the data to be stored in.

### *Constructors*

A constructor is a special method of instantiating an object. Programmers use constructors because when one is called when an object of a class is created, it can be used to set initial values for fields.

```

using System;

class animal // animal class
{
    public string color;           // field
    public int eyes;               // field
    public string animalType;      // field

    public animal(string animalColor, string animalAnimalType) //animal class
    constructor with multiple parameters
}

```



```

{
    color = animalColor;

    animalType = animalAnimalType;
}
}

public class Program
{
    public static void Main(string[] args)
    {
        animal dog = new animal("brown", "dog");

        Console.WriteLine(dog.color);           //brown
        Console.WriteLine(dog.animalType); //dog
    }
}

```

The above example constructor assigns the arguments passed to it at creation of the object to the animal class fields.

Ei `new animal("brown", "dog");` is `color = "brown"; animalType = "dog";` in the animal class. This is a much cleaner way to give individual objects different values for fields at instantiation.

#### With Constructor

```

class animal // animal class
{
    public string color;           // field
    public int eyes;               // field
}

```

#### Without Constructor

```

class animal // animal class
{
    public string color;           // field
    public int eyes;               // field
    public string animalType;     // field
}

```



```

public string animalType;           // field

public animal(string animalColor, string
animalAnimalType) //animal class constructor
with multiple parameters
{
    color = animalColor;
    animalType = animalAnimalType;
}
}

public class Program
{
    public static void Main(string[] args)
    {
        animal dog = new animal("brown", "dog");
        animal cat= new animal("orange", "cat");
        animal bat= new animal("black", "bat");
    }
}

```

```

public class Program
{
    public static void Main(string[] args)
    {
        animal dog = new animal();
        dog.color = "brown";
        dog.animalType = "dog";

        animal cat = new animal();
        dog.color = "orange";
        dog.animalType = "cat";

        animal bat = new animal();
        bat.color = "black";
        bat.animalType = "bat";

    }
}

```

As you can see above, the constructor makes it easier to create objects and assign their fields with values during creation.

A few things to note about constructors. They must have the same name as the class they construct objects for, and they must not contain a return type in their declaration.

### *Inheritance*

C# allows classes to inherit fields and methods from another class where the base (parent) class is the class being inherited from, and the derived (child) class is the class that inherits



from another. To inherit from a class the : (colon) symbol is used, with the parent class on the right-hand side.

```
class animal                                // base (parent) class
{
    public string color;                    // parent field
    public void sound(string sound)        // parent method
    {
        Console.WriteLine($"The sound I make is: {sound}!");
    }
}

class dog : animal // derived (child) class
{
    public string breed; // child field
}

public class Program
{
    public static void Main(string[] args)
    {
        dog fudge = new dog(); // create new dog class object called fudge
        fudge.color = "brown"; // brown
    }
}
```

To prevent classes from being able to inherit from a class, use the sealed keyword on the class you want to protect.

```
sealed class animal
```



```

{
    ...
}

class dog : animal
{
    ... // error will occur as cannot inherit from 'sealed' animal
}

```

Note: if a class constructor is inheriting from another class constructor, you must also satisfy the parameters of the parent class.

Take for example the below constructors for creating points on a map.

```

class Point
{
    public int X;
    public int Y;
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}

class MapLocation : Point
{
    public MapLocation(int x, int y) : base(x, y)
    {}
}

```



Here, when a map location object is created, a point object is created too. They are not two distinct objects - a map location is also a point.

The point constructor requires two parameters ( x and y coordinates) to create object, and so when a map location object is created it first needs to call the point class constructor and pass it its parameters. That's why above, the map location constructor also takes x and y coordinates. To tell it to call the point class constructor and pass the x and y parameters we typed : base followed by a pair of opening and closing parentheses which list the parameters or arguments we want to pass to the base constructor (parent class constructor). So what the above map location constructor is doing is taking x and y parameters, then immediately passing these to the point constructor, where the point constructor then assigns these to the corresponding fields in its constructor method.

Note: C# doesn't support multiple inheritance, instead you can have a class implement multiple interfaces.

### *Polymorphism*

Polymorphism (poly = many, morphism = forms) occurs when we have many classes that are related to each other by inheritance. Polymorphism uses inherited methods to perform different tasks.

Take for instance our animal example again, where our animal class has a method called sound, which takes the string parameter of sound. It is currently dynamic as it stands:

```
using System;

class animal                                // base (parent) class
{
    public string color;                    // parent field
    public void sound(string sound)        // parent method
    {
        Console.WriteLine($"The sound I make is: {sound}!");
    }
}

class dog : animal // derived (child) class
```



```

{
    public string breed; // child field
}

class bear : animal // derived (child) class
{
    public string species; // child field
}

public class Program
{
    public static void Main(string[] args)
    {
        dog fudge = new dog(); // create new dog class object called fudge
        fudge.color = "brown"; // assigns color field the value of brown
        Console.WriteLine(fudge.color); // brown
        fudge.sound("woof"); // assigns parent method sound value of 'woof'
        fudge.sound("bork"); //The sound I make is: bork!

        dog terry = new dog();

        terry.sound("oof"); //The sound I make is: oof!

        bear pudgy = new bear();

        pudgy.sound("grr"); //The sound I make is: grr!
    }
}

```



```
}
```

But what about for methods that don't take a parameter, who's output you do want to change for every subclass? This is where polymorphism comes in, with its keywords `virtual` and `override`.

Without specifying `'virtual'` in the superclass method, and `'override'` in the child classes methods, the base class method will override the derived class methods that share the same name, and output only the base class output. ei , without using the polymorphism keywords below, all our object outputs would be `All animals make a noise!`.

```
using System;

class animal // base (parent) class
{
    public string color; // parent field

    public virtual void noise() // parent method
    {
        Console.WriteLine("All animals make a noise!");
    }
}

class dog : animal // derived (child) class
{
    public string breed; // child field

    public override void noise() // child override of parent method
    {
        Console.WriteLine("Dogs go woof!");
    }
}
```



```

class bear : animal // derived (child) class
{
    public string species;          // child field

    public override void noise()    // child override of parent method
    {
        Console.WriteLine("Bears go grr!");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        dog fudge = new dog(); // create new dog class object called fudge
        fudge.noise(); //Dogs go woof!

        dog terry = new dog();
        terry.noise(); //Dogs go woof!

        bear pudgy = new bear();
        pudgy.noise(); //Bears go grr!
    }
}

```



## Abstraction

An abstract base class is a base class that defines what it means to be an object of that class, but isn't actually a class to which an object can be created.

Abstraction can be achieved with either abstract classes or interfaces, and consists of abstract classes - which are restricted classes that cannot be used to create objects and to access code within it, it must be inherited from another class- and abstract methods - which can only be used in an abstract class, and do not contain a body, as the body is provided by the derived class.

Continuing on with the previous animal example, we can make an object from the animal class.

```
animal anAnimalObject = new animal();  
  
anAnimalObject.noise(); //All animals make a noise!
```

But what if we only wanted the animal class to hold methods, properties and fields that derived classes can assign values to, to create unique objects of their own class, and not have any object to be able to be created from the animal class itself? That's where the abstract keyword comes into play.

See the below example now. There are two important things to note about the example.

1. The 'abstract' keyword is not valid on fields. Therefore you define fields as private, and create properties to be able to access and or edit them.
2. Abstract methods cannot be marked as 'virtual', instead we just ensure that the derived class has the 'overrides' keyword.

```
using System;  
  
abstract class animal // base (parent) class  
{  
    private string color; // parent "abstract" field  
    public string Color {get;set;}  
  
    private bool wings;  
    public bool Wings {get;set;}}
```



```

    public abstract void noise();                // parent abstract method

    public virtual void isAnimal()              // parent regular method
    {
        Console.WriteLine("I am an animal!");
    }
}

class dog : animal // derived (child) class
{
    public string breed;                        // child field

    public override void noise()              // child override of abstract parent method
- child provides body to method
    {
        Console.WriteLine("Dogs go woof!");
    }

    public override void isAnimal()           // child override of parent method
    {
        Console.WriteLine("I am a dog!");
    }
}

class bat : animal // derived (child) class
{
    public override void noise()              // child override of parent method
    {

```



```

        Console.WriteLine("Bats go Sqree!");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        dog fudge = new dog(); // create new dog class object called fudge
        fudge.noise(); //Dogs go woof!

        dog terry = new dog();

        terry.isAnimal(); //I am a dog!

        bat angel = new bat();

        angel.noise(); //Bats go Sqree!

        angel.Wings = true;

        Console.WriteLine(angel.Wings); //true
    }
}

```

If we now tried to create an object from the animal class we'd throw errors.

```

animal animalObject = new animal();

```

Cannot create an instance of the abstract class or interface 'animal'



## Interfaces

Interfaces are another way to instigate abstraction. An interface is a completely abstract class, in which it can only contain abstract methods and properties. To create an interface, use the interface keyword followed by a name for the interface and a pair of curly braces. The curly braces will contain all the interface methods and properties.

Programmers typically name interfaces with a starting capital I (for interface), as this helps differentiate between abstract classes, interfaces and classes.

```
interface Ianimal
{
    string Color {get;set;}

    bool Wings {get;set;}

    void noise();

    void isAnimal();
}
```

See our previous abstract class example below, now changed into an interface.

There are a few important things to note about the example.

1. Interfaces cannot contain fields.
2. By default, members of an interface are abstract and public (notice we do not write these in the example, as it's not necessary to).
3. Methods must have empty bodies in an interface definition.
4. You do not use the override keyword when implementing an interface method.
5. On implementation of an interface, you must override all of its methods.
6. An interface cannot contain a constructor.

```
using System;

interface Ianimal
{
    string Color {get;set;}

    void noise();
}
```





```

        void isAnimal();
    }

class dog : Ianimal
{
    private string _color;

    public string Color
    {
        get => _color;
        set => _color = value;
    }

    public void noise()
    {
        Console.WriteLine("Dogs go woof!");
    }

    public void isAnimal()
    {
        Console.WriteLine("I am a dog!");
    }
}

public class Program
{
    public static void Main(string[] args)

```



```

{
    dog fudge = new dog(); // create new dog class object called fudge

    fudge.noise(); //Dogs go woof!

    fudge.isAnimal(); //I am a dog!

    fudge.Color = "brown";

    Console.WriteLine(fudge.Color); // brown
}
}

```

Note: we named the private field in dog `_color`, as it is typical naming convention to name private fields with an underscore as the first character.

An interface pre-defined in the System namespace is `Comparable`. Each class that implements it, is free to provide it's own custom comparison logic inside the `CompareTo` method.

```

interface Comparable
{
    int CompareTo(object obj);
}

```

### *Multiple Interfaces*

To implement multiple interfaces, separate them with a comma.

```

class Example : IExampleOfConvention, IExampleOfMultiInterface{

    //

}

```

### Namespaces

Namespaces are groupings of similar classes. They provide a "named space" in which the application resides. The purpose of namespaces is to prevent conflict in classes names. For



example, without namespaces you wouldn't be able to make a class named Console, as the .NET framework already uses one in its pre-existing System namespace.

Here's another example:

```
namespace anExample {  
  
    class System {}  
  
    class program{  
  
        static void Main(string[] args){  
  
            System.Console.WriteLine("Welcome to C#!");  
  
        }  
  
    }  
  
}
```

The above outputs the error `'System' does not contain a definition for 'Console'` this is because C# doesn't look at the global system namespace that exists in the .NET framework as it's now looking at our system class instead. And our system class indeed doesn't contain a definition for console. One way to mitigate this issue is to tell the interpreter to look at the global scope rather than our one here. We do this by adding the keyword `global` followed by two colons.

```
global::System.Console.WriteLine("Welcome to C#!");
```

Note: it's not recommended to create your own name spaces that use pre-existing names in the .NET framework, as it can lead to confusion and potential errors like above.

Another example of class name conflict mitigation with namespaces:

```
namespace anExample {  
  
    class program{  
  
        static void Main(string[] args){  
  
            System.Console.WriteLine("Welcome to C#!");  
  
        }  
  
    }  
  
}
```



```
namespace myNamespace{

    class program {

        //code

    }

}
```

The class 'program' in myNamespace doesn't conflict with the class 'program' in the anExample namespace because they exist in different scopes. If we wanted to reference the class 'program' in myNamespace, we simply call it by typing the namespace name dot(.) the name of the class.

```
using System;

namespace anExample
{

    public class program

    {

        public void hello()

        {

            Console.WriteLine("Welcome to C#!");

        }

    }

    namespace myNamespace // myNamespace is now a nested namespace within
anExample

    {

        public class program

        {

            public void greeting()
```



```

        {
            Console.WriteLine("I am also in a class called program.");
        }
    }
}

public class scopeExample
{
    public static void Main(string[] args)
    {
        // creates an object of the anExample program class
        anExample.program outer = new anExample.program();
        outer.hello(); //Welcome to C#!

        // creates an object of the nested myNamespace program class
        myNamespace.program inner1 = new myNamespace.program();
        inner1.greeting(); //I am also in a class called program.
    }
}
}

```

Namespaces are also useful because writing the name of a namespace over and over prepended onto the type name can clutter code, C# offers a shortcut with the using directive.

Take the prerequisite C# example for instance, if we were to remove the 'using System' statement, then for every 'Console.WriteLine' statement we would have to prepend 'system.' to it, and our code would quickly become hard to read, especially with more complex code.

```
var Fruit = "apple";
```



```

var FRUIT = "apple";

var fruit = "apple";

System.Console.WriteLine(FRUIT);

System.Console.WriteLine(Fruit);

System.Console.WriteLine(fruit);

```

```

using System;

namespace fruits
{
    ...

    var FRUIT = "apple";

    ...

    Console.WriteLine(FRUIT);

} }

```

### Create A Namespace

You can create your own namespaces by declaring the namespace keyword, followed by the name of your namespace and a pair of curly braces. Within these curly braces, you place your code, ei classes, fields, ect.

```
namespace myNamespace {}
```

To create nested namespaces you can use on of the two methods shown below.

```

namespace anExample
{
    namespace myNamespace
    {
        //code
    }
}

```



```
}  
  
}
```

```
namespace anExample.myNamespace  
  
{  
  
    //code  
  
}
```

## Design Patterns

Design patterns are elegant, adaptable, and reusable solutions to everyday software development problems. These design patterns are ways in which code can be written / structured in order to solve a particular problem or scenario.

### Characteristics Of A Design Pattern

- Reusable
- Contextual
- Useful Solution

### *Categories*

There are 3 common types of design patterns. These are creational, structural and behavioural.

### *Creational*

Creational design patterns are patterns that control the mechanism for creating objects. They try to create objects in a manner suitable to the situation. Examples of this pattern are shown below.

### Singleton

Used to control instantiation of an object. It ensures that only one instance is allowed at any one time.

Singleton involves a single class which provides a way to access its only object - that can be accessed directly without need to instantiate it.

To create a singleton, we create the constructor as private and have a static instance of itself.

```
using System;
```



```

public class Program
{
    public static void Main()
    {
        //ask user their name and save it in the name variable

        Console.WriteLine("What is your name?");

        string name = Console.ReadLine();

        //Get the only object available

        SingletonExample newObject = SingletonExample.GetObject();

        //show the message and pass argument

        newObject.greeting(name);
    }
}

public class SingletonExample {

    //create an object of the SingletonExample class

    private static SingletonExample myObject = new SingletonExample();

    //make the constructor private so that this class cannot be instantiated

    private SingletonExample(){}
}

```





```

//Get the only object available

public static SingletonExample getObject(){

    return myObject;

}

//class method called greeting

public void greeting(string name){

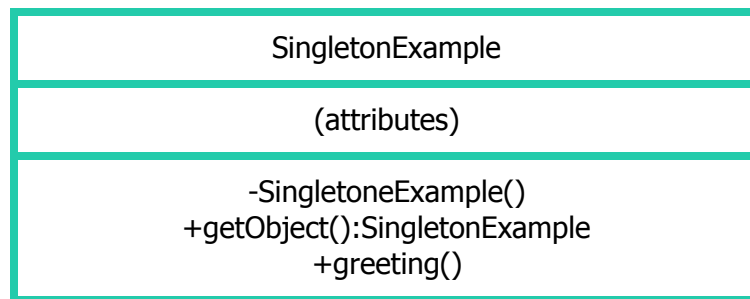
    Console.WriteLine("Hello " + name + "!");

}

}

```

Singleton's can be represented by the UML class model.



A real life example of a singleton pattern would be: A mobile application to log in and perform banking operations for bank customers. For security purposes, only a maximum of one authorised connection is allowed at any one time for each account.

### Factory

The factory design pattern is used to create a factory object. This is used to instantiate different objects at runtime. This is why it is referred to as a factory design as it acts like a 'factory' in that you can ask it to create various types of objects 'on the fly'.

This pattern is used because:

- You cannot anticipate the type of objects you need to create beforehand.
- A class requires its subclasses to specify the objects it creates.
- You want to localize the logic to instantiate a complex object.

With this pattern, you create objects without exposing the creation logic to the client and refer to newly created objects using a common interface.

```
using System;
```



```

public class Program
{
    public static void Main()
    {
        Factory myNewAnimalObj = new Factory();

        //get an object of dog and call its myAnimal method.
        Ianimal terry = myNewAnimalObj.getAnimal("DOG");

        //call the method of terry
        terry.myAnimal();

        Ianimal pudgy = myNewAnimalObj.getAnimal("bear");
        pudgy.myAnimal();
    }
}

public interface Ianimal {
    void myAnimal();
}

public class dog : Ianimal {
    public void myAnimal() {
        Console.WriteLine("I am a dog!");
    }
}

```



```

    }
}

public class bear : Ianimal {

    public void myAnimal() {

        Console.WriteLine("I am a bear!");

    }

}

public class Factory {

    //use getAnimal method to get object of type Ianimal

    public Ianimal getAnimal(String type)

    {

        switch (type.ToLower())

        {

            case "dog":

                return new dog();

                break;

            case "bear" :

                return new bear();

                break;

            default:

                return null;

        }

    }

}

```



```
}  
  
}
```

Similar to singleton, you can represent factories using class models.

### Builder

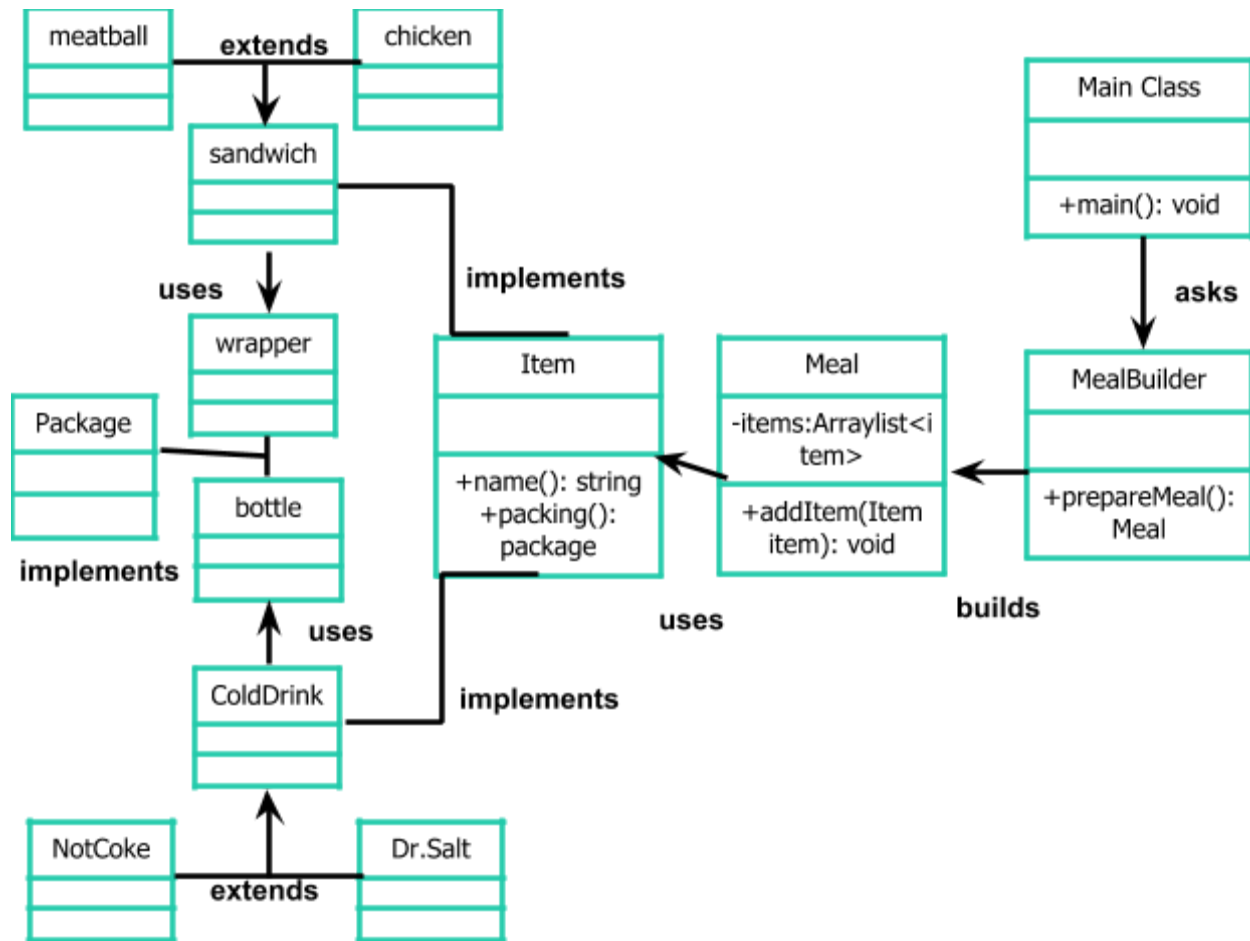
A builder pattern builds a complex object by using simple objects and a step by step approach.

Take for example, a fast food restaurant called Underpass. A typical meal would be a sandwich and a cold drink. The sandwich could be either a meatball or chicken sandwich and will be packaged in a wrapper. And the cold drink could be either a Notcoke or Dr.Salt and will be packaged in a bottle.

To build this in a builder pattern we would create an Item interface representing the available food items of burgers and cold drinks, and a Packing interface representing the available packaging of food items. We would also create classes implementing the Packing and Item interfaces as sandwiches would be packaged in wrappers and cold drinks would be packed in bottles.

We would then create a Meal class which would have an ArrayList of Item and a MealBuilder to build different types of Meal objects by combining different Item objects. Our main class will use MealBuilder to build a Meal.





## Structural

Structural design patterns are patterns concerned with the composition of objects, and the relationships and interaction between objects.

### Bridge

The bridge pattern decouples implementation class and abstract classes by providing a bridge structure between them. This pattern involves an interface which acts as a bridge to make the functionality of concrete classes independent from the interface implementer classes. Resulting in both types of classes being able to be altered structurally without affecting each other.

### Composite

This pattern is used to represent part-whole hierarchies of objects. In object-oriented programming, a composite is an object with a composition of one+ similar objects, where each of these objects has similar functionality. The composite pattern is common in tree-structured data.



## Decorator

This pattern acts as a wrapper to an existing class, as it allows a user to add new functionality to an existing object without altering its structure. It essentially allows an extension of a class or additional functionality to be added to a class dynamically without altering existing class structures.

## Adapter

Also known as a 'Wrapper' or 'Translator' pattern. The adapter pattern allows one type of interface to communicate with another. It essentially adapts an 'adaptee' object, making it fully accessible to another object of a different type.

## *Behavioural*

Behavioural design patterns are patterns that deal with the behaviour of objects and the communication between objects.

## State

A state pattern allows an object to change its behaviour depending upon its internal state.

In the state pattern, we create objects which represent various states and a context object whose behavior varies as its state objects change.

An example of this pattern would be a company's pool car having options of available, in use, booked and being serviced.

This pattern can be demonstrated using class diagrams.

## Strategy

In strategy patterns, a class behavior or its algorithm can be changed at run time. To implement a strategy pattern, you create objects which represent various strategies and a context object whose behavior varies as per its strategy object.

# Programming Principles

Code reusability is one of the core programming principles in any language. It aims to save time and resources, and reduce redundancy by taking advantage of assets that have already been created. It enforces the DRY principle. Which stands for Don't Repeat Yourself. The goal is that any modification to a single element in a system does not require a change in other logically unrelated elements.



Violations of the DRY principle are referred to as WET which stands for either Write Everything Twice, We Enjoy Typing, or Waste Everyone's Time.

Maintainability is another core programming principle. And it is simply how easy it is to maintain your code. Maintaining code refers to things like making changes, bug fixes and refactoring - which is the process of improving code efficiency and cleanliness.

## Glossary

| Word Or Phrase      | Definition   |
|---------------------|--|
| Acceptance Testing  | is a level of software testing where a system is tested for acceptability. The purpose is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.                                       |
| Activity Diagrams   | are graphical representations of workflows and actions with support for choice, iteration and concurrency.   |
| Actors              | In software development, a use case is a list of actions or event steps typically defining the interactions between a role (an actor) and a system to achieve a goal. The actor can be a human or other external system.                             |
| Agile               | An umbrella term for many different development methodologies. It is characterised by the division of tasks into short phases of work and frequent reassessment and adaptation of plans.   |
| Backlog             | An accumulation of uncompleted work or matters needing to be dealt with.   |
| Big Bang Deployment | is the instant changeover, when everybody associated with the old system moves to the fully functioning new system on a given date.  |
| Bug-fix             | A correction to a bug in a program or system.  |
| Business Analyst    | is someone who analyses an organisation and documents its business, processes or systems, assessing the business model or its integration with technology.   |
| Business Case       | A justification for a proposed project based on its expected commercial benefit.   |
| Business Drivers    | is a resource, process or condition that is vital for the continued success and growth of a business.  |
| Client GUI          | A graphical user interface (GUI) is a human-computer interface (i.e., a way for humans to interact with computers) that uses windows, icons and menus and which can be manipulated by a mouse (and often to a limited extent by a keyboard as well). |



|                                |   |
|--------------------------------|---|
| Component Diagram              | show the dependencies and interactions between software components.   |
| Cross Site Scripting           | is a type of computer security vulnerability typically found in web applications. XSS enables attackers to inject client-side scripts into web pages viewed by other users.   |
| Data Flow Models               | is a diagrammatic representation of the flow and exchange of information within a system.   |
| Defensive Programming          | is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances.   |
| Denial Of Service              | An interruption in an authorised user's access to a computer network, typically one caused with malicious intent.   |
| Development Methodologies      | refers to the framework that is used to structure, plan, and control the process of developing a system.  |
| DevOps                         | Is a software development process that emphasises communication and collaboration between software developers and operations.   |
| Distributed Denial Of Service  | The intentional paralysing of a computer network by flooding it with data sent simultaneously from many individual computers.   |
| Domain Experts                 | Domain expert is a person who is an authority in a particular area or topic.  |
| Entity                         | A thing with distinct and independent existence.  |
| Entity Relationship Models     | (ERM) is a theoretical and conceptual way of showing data relationships in software development.  |
| Evolutionary Prototyping Model | This prototype evolves into the final product. The main goal is to build a very robust prototype in a structured manner and then refine it. The prototype forms the heart of the new system and the improvements and further requirements will then be built. |
| Extreme Programming            | (XP) is a software development methodology which intends to improve software quality and responsiveness to changing customer requirements.  |
| Feasibility Phase              | Feasibility is the measure of how well a proposed system solves the problems.   |
| Feasibility Study              | An assessment of the practicality of a proposed plan or method.   |
| Functional Requirements        | In software engineering a functional requirement defines a function of a system or its component. A function is described as a set of inputs, the behaviour, and outputs.   |
| Malware                        | Software which is specifically designed to disrupt, damage, or gain authorised access to a computer system.   |
| Manual Testing                 | is the process of manually testing software for defects. It requires a tester to play the role of an end user.  |
| Non Functional Requirements    | is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviours.  |
| OOP                            | Stands for Object Oriented Programing   |





|                             |  |
|-----------------------------|--|
| Product Owner               | Part of the product owner responsibilities is to have a vision of what they wish to build, and convey that vision to the SCRUM Team.   |
| Project Manager             | The person in overall charge of the planning and execution of a particular project.  |
| Prototypes                  | A first or preliminary version of something from which other forms are developed.  |
| Prototyping                 | is the activity of creating early sample versions of software. Typically simulation only a few aspects of the final product. There are two models for prototyping, throwaway model and evolutionary model.   |
| Quality Control             | A system of maintaining standards by testing a sample of the output against the specification.   |
| Requirements Analysis       | Also called requirements engineering, is the process of determining user expectations for a new or modified product. These features,(requirements) must be quantifiable, relevant and detailed. These requirements are often called functional specifications. |
| Requirements Documentation  | (PRD) is a document containing all the requirements to a certain product. It is written to give people an understanding in what a product should do.   |
| SCRUM                       | SCRUM is a methodology that allows a team to self-organise and make changes quickly, in accordance with agile principles.  |
| SCRUM Master                | A SCRUM master is the facilitator for an agile development team. They manage the process for how information is exchanged.   |
| Secure Development          | Secure development is a practice to ensure that the code and processes that go into developing applications are as secure as possible.   |
| Social Engineering          | The use of deception to manipulate individuals into divulging confidential or personal information that may be used for fraudulent purposes.   |
| Software Architecture       | Software architecture is the structure of structures, of a system consisting of entities and their properties, and the relationships among them.   |
| Software Designer           | A Software designer is responsible for the process of implementing software solutions to one or more sets of problems.   |
| Software Developer          | Develops software. Other job titles used with similar meanings are programmer, software analyst, and software engineer.  |
| Software Release Engineer   | concerned with the compilation, assembly, and delivery of source code into finished products or other software components.   |
| SQL                         | SQL is an abbreviation for structured query language. SQL is a standardised query language for requesting information from a database.   |
| SQL Injection Vulnerability | SQL injection is a code injection technique, used to attack data-driven applications, in which nefarious SQL statements are inserted into an entry field for execution.  |
| Systems Architect           | Systems architects define the architecture of a system in order to fulfil certain requirements. Such definitions include: a breakdown of the system into components, the component interactions and interfaces, and the  |



|                             |  |
|-----------------------------|--|
|                             | technologies and resources to be used in the design.   |
| Technical Architect         | Responsible for defining the overall structure of a program or system, overseeing IT assignments that are aimed at improving the business, and ensuring all parts of the project run smoothly. |
| Test Plan                   | A test plan is a document detailing the objectives, target market, internal beta team, and processes for a specific beta test for a software or hardware product.                              |
| Test Script                 | is a set of instructions that will be performed on the system under test, to test that the system functions as expected.   |
| Throwaway Prototyping Model | This prototype is thrown away once it is has been reviewed. They are quick to produce, and often visually resemble the final product but have no to little functionality                       |
| UML                         | Unified Modelling language (UML) is a standardised modelling language enabling developers to specify, visualise, construct and document artefacts of a software system.                        |
| Unit Testing                | Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinised for proper operation.      |
| Wireframes                  | A skeletal model in which only lines and vertices are represented.   |