Programming Paradigms

- A paradigm simply means a way of thinking.
- Programming paradigms include procedural, object orientated, assembly, functional and declarative.
- Most programming languages and paradigms are said to be “Turing complete.”
- This means that a language is able to solve all the problems a computer is able to solve.
- Some problems are better suited to being solved in a certain paradigm.
- This makes languages written in certain paradigms better suited to solving certain problems.

Procedural

- One of the most common procedural paradigms in use.
- Sometimes referred to as imperative languages. This means to give an order.
- Tells a computer what to do step by step.
- Includes:
  - Sequence
  - Selection
  - Iteration
  - Recursion
  - Data types
  - Data structures
  - Abstract data structures such as stack and queues
- It uses modular techniques to split large problems into manageable chunks.

Assembly Language

- Assembly languages are known as low level languages.
- Low level instruction - opcode + operand.
- Each type of processor has its own unique instruction set of low level assembly codes available to it.
- This means that an assembly level language written for one processor will not work on another.
- As assembly language is translated directly into machine code, these instructions sets allow us to represent the actual binary 1s and 0s by using short codes known as mnemonics.
- This is so we don't actually have to remember or code the actual binary values themselves.

Functional

- Languages such as Hascal.
- Programs consist of functions that accept data and return data.
- Used in big data, the concept of manipulating huge amounts of data.

Declarative

- Supported by languages such as SQL, where you write statements that describe the problem to be solved, and the language implementation decides the best way of solving it.
- SQL is a standard language used to read/write from/to databases.
- SQL is a declarative language (this means that you tell it what you want it to do, you don't have to tell it how to do it).
- In SQL, you write statements declaring what we want from the database.

Object Orientated Programming (OOP)

Object Orientated Languages

- Object oriented programming relies on objects in the real world being represented or classified and then every object identified being placed in its own class.
- A class is like a template - we use it to set out and define what attributes and methods an object of a certain type should have.
- This class is not an actual object itself, it's just a cutting tool to make sure that every object we make from that class looks the same.
- This is one of the major benefits of OOP.
- When we want to create an actual copy or an “instance” of a class we can use a single line of code to create it using the word “new.”
- This in OOP is called instantiation: “The process of creating an object from a class template.”
- Every language which implements OOP has its own syntax and way of doing it.

Object Orientated Programming

In OOP, you can use inheritance to quickly and easily reuse code and then extend the attributes and methods without affecting the original

code.

- For example, if you needed to make an employee class, an employee is still a person, but requires more information such as an NI number.
- You do not rewrite all the code, we simply create a class that inherits all the details of the person class and then adds the extra attributes and methods needed. This results in a class and a subclass.
- Overriding happens when a method in a subclass shares the same name as a method further up the object tree.
- The overridden method takes precedence and replaces the method inherited form the superclass.
- Using the super prefixes overrides overriding.

Encapsulation

The protection of the attributes and methods of an object so that they can't be accessed or altered by other objects.

This concept allows you to completely protect the data associated with each object.

It can't be accidentally altered by another part of the program without using the code you provide thus this keeps you, the programmer, in control.

- Other objects (instantiations of a class) are prevented from directly accessing the data of attributes inside an object other than by going through or using the method that are provided.
- In OOP, we use the keyword private to show that the following attribute is only accessible from within the class.
- Any attempt to access or change a private attribute outside of the class will result in error.
- The method must be used instead.
- This now means that we must make sure to provide a method in the class in order to allow other parts of the program to alter one of the private attributes.
- This method therefore has to be public.
- Because it is public, it can be called by other parts of the program and thus it will be able to change the internal, private attribute.
- However, you now have complete control as you're the one writing the code in that method which means the attributes can only be changed in the way you specify.

Polymorphism

- Polymorphism makes sure that when a symbol or function have different meanings the right meaning is chosen based on the context.

Polymorphism

Giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.

- Polymorphism literally means “something that occurs in several different forms.”
- There are two main types in OOP:
    - Static
    - Dynamic

Static Polymorphism

- Static polymorphism allows you to implement multiple methods of the same name (but with different parameters) within the same class.
- This process is known as method overloading.
- The parameter sets must differ in at least one of these ways.

Dynamic Polymorphism

- Within an inheritance hierarchy, a subclass is able to override a method of its superclass - this allows the coder of the subclass to customise or completely replace the behaviour of that method.
- This is a form of polymorphism - both methods share the same name and parameters, but they provide different functionality depending on whether they’re implemented by the superclass or the subclass.

Development Methodologies

- When writing a program, you are going through a number of distinct problems.
- Together, these phases are known as the Systems Development Life Cycle (SDLC):

Phase 1: Feasibility Study

- Focuses on helping ask the essential question: should we proceed?

Phase 2: Requirements

- Analyse carefully what the user wants and we define the project goals into a set of requirements.
- Try to gain as much information as possible.
- Analyse carefully what the client wants / user needs.
- Defining the procedures or what the system needs to do.
- Breaking down the problem into main functions.
    - This involves defining the data, uses, volumes and characteristics.
- The future - development plans and expected growth rates.
- Problems with any existing systems.
- The end result of this phase is a requirement specification document, which outlines everything the user wants the proposed system to do.
- It is this document which is checked and signed off during acceptance testing at the end of a project.

Phase 3: Analysis and Design

- Describes the desired features and operations in detail.
- It includes mocked up screen layouts, business rules, process diagrams, pseudo code and lots of other documentation.
- **Design:**
    - Breaking down the process into individual modules and further breaking these down until each module performs a well designed task.
    - Things considered may include: processes, user interface, output, input, data structures (Arrays, Lists, DBs etc).
    - Security.

Phase 4: Implementation

- This is the phase where the actual code is written.

Phase 5: Testing

- All the code has to be placed in a special testing environment to be tested for errors and bugs.
- This phase checks that the system actually works as intended.
- **Acceptance Testing: a testing technique performed to determine whether or not the software system has met the requirement specifications. The main purpose of this test is to evaluate the system's compliance with the business requirements and verify if it has met the requirements specified earlier.**

Phase 6: Evaluation

- Using criteria, the finished system is evaluated against the objectives which were originally agreed between the client and the analyst.
- Evaluation includes:
    - Evidence of success or not: did the system meet the requirements?
    - Evidence that the new system functions as designed, both in terms of algorithmic functions and interface design.
    - Post-implementation review, which is a critical examination of the system three to six months after it has been put in operation.
    - The solution should be evaluated on the basis of effectiveness, usability and maintainability

Phase 7: Maintenance

- Covers everything that happens in the rest of the software's life, after it has been installed. Changes, corrections, additions and updates are all included.
- Discuss the future maintenance of the program and any limitations in the current versions.

Development Models

Waterfall Model

- Derives its name due to the cascading effect of one phase to another.
- Each phase has a well defined starting and end point, with identifiable deliveries onto the next phase. (each step has specific outputs that lead into the next step).
- Each step is completed one at a time from beginning to end.

- It is possible to return to a previous stage if necessary but the model shows that the developers then have to work back down through the following stages.
- The user / customer is involved at the start of the process, in the analysis stage, but then has little input until the evaluation stage.
- It has now been superseded by more effective models.

Rapid Action development (RAD)

- Methodology for designing and writing software that includes producing successive prototyping versions of the software until the final version has been produced.
- Since very large projects may be developed over a long period of time during which both technology and user requirements change.
- Major changes at late stages of development can sometimes lead to projects being cancelled or restarted, at a considerable cost.
- In response to this problem the RAD methodology was introduced, offering the promise of much faster completion of major projects.
- The ideas behind it include:
  - Workshops and focus groups to gather requirements rather than formal requirements document.
  - The use of prototyping to continually refine the system in response to user involvement and feedback.
  - Producing within a strict time limit each part of the system, which may not be perfect but which is good enough.
  - Reusing any software components which have already been used elsewhere.
- Iterative development and each cycle can last a week.

Spiral Model

- Risk driven approach to development. It acknowledges that risk is at the heart of many large scale development projects. It is designed to take into account these risks while they arise in the project and then deal with them before they become a major problem.
- At the start of the process the requirements are defined and the developers working towards an initial prototype.
- Each successive loop around the spiral generates a refined prototype until the product is finished.
- The model has four states:
  - The first quadrant determines objectives.
  - The next quadrant identifies and resolves risks.
  - A prototype is then developed, which is tested in the third quadrant.
  - The final quadrant is either completion or planning the next iteration.

Agile Methodologies and Extreme Programming

- A group of development methodologies.
- Used when stages of software development may not be completed in a linear sequence.
- The developer may then go back to design another aspect of the system.
- Focus on the idea that the requirements will constantly shift and change whilst the development is taking place.
- This can only be done by producing software in an iterative manner, with each iteration having increased requirements and being shown to the user.
- Throughout the process, feedback will be obtained from the user, this an iterative process during which changes made are incremental as the next part of the system is built.
- **Extreme programming** is an example and it has short iterations.
- It is intended to improve software quality and responsiveness to changing customer requirements.
- It is a type of agile software development in which frequent “releases” of the software are made in short development cycles.
- This is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.
- For this to work, a company will often embed a user into the development team, so it will be able to give instant feedback on the next iteration.

Development Methodologies Pros & Cons

Waterfall

Aa Pros	Cons
Simplicity	Lack of flexibility
Easy model to manage	High risk factor
Each stage has clear deliverables	End user is very removed from development
Each stage has well defined boundaries	Poor choice for systems with unsure requirements
Easy to spot timescales slips	

Rapid Action Development

Aa Pros	Cons

User feels involved	Regular contact with client needed
End product is more likely to match user's requirements	Does not scale well to large products
Untitled	Not suitable if efficiency of code is priority

Spiral Model

Aa Pros	Cons
Focuses on risks	Risk management is a very specialised and costly skill
Works well for large products	

Agile and Extreme

Aa Pros	Cons
Quality of code is very high	Increased development cost
Efficient code and few bugs	Heavy collaboration required
Untitled	Embedded end user required

Testing Strategies

- Programs need to be tested to make sure that they are functional.
- There are many types of testing strategy. All these tests have the same goals, but are different in how and when they work.

Black Box Testing

- Black box testing is only concerned with the inputs and outputs of the program and not with how the program works.
- You are simply testing whether the inputs produce the outputs that would be expected.
- It does not consider the efficiency of the program. Ideally, you would want to test each possible input for their output.

White Box Testing

- White box testing techniques test the actual algorithm in the code to make sure all parts of the algorithm are functioning as intended.
- The technique identifies every possible route or path of execution through a program.
- On each test run the path of execution is noted so it can be compared with other runs.
- This technique actually cares about how efficiently the program is run.

Alpha and Beta Testing

- Alpha and beta testing are carried out when the software is nearly complete and being tested as a whole.
- They apply especially to commercial software such as computer games.
- The only difference between alpha and beta testing is the stage in which it takes place and who performs the testing.
- Alpha testing happens sooner and is normally limited to being tested by the internal employee’s and the friends and family of the company.
- It is usually a very early version of the software and quite full of bugs.
- Beta testing tends to be tested with a wider community through beta signup programs.
- The program now is in an almost finished state, and the developers are usually wanting to test areas such as load balancing and what happens when a program is tested by ten thousand people at once.

Test Data and User Feedback

Types of Test Data

- Valid data - data which you would normally expect a user to input.
- Invalid or extreme data - data which should generate an error message if it was an input.
- Borderline or boundary data - you need to be especially careful to test data around the boundaries between different cases to ensure they are dealt with correctly.

Translators

- Any program which converts source code into machine code.
- There are three different types:
  - Interpreters
  - Compilers
  - Assemblers

Assemblers

- **Assembly Language** is a low level language:
  - It uses mnemonic code rather than 1s and 0s.
  - It maps one to one with machine code.
  - It is easier to understand and read than binary code.
- Responsible for converting low level assembly language directly into object code.
- Each line of code is converted or looked up by the assembler directly into one line of binary equivalent machine code.
- The assembly code for one processor may not work for another, meaning it is not that portable.

Compilers and Interpreters

- Responsible for converting high level language into object code or machine code.
- Both have a one to many relationship (each high level line translates into many lines of binary equivalent machine code.)

Interpreters

- **Interpreters** take one line of high level code, convert it into machine code and then run it.
- This means the CPU will execute an interpreted instruction and then the interpreter will move on to the next instruction and so on.
- These are useful as they can stop immediately at the first line where there is an error, however it requires access to the whole source code each time it needs to translate it.
- This means that it is quite a slow translator.
- *Translation happens at run time.*

Compilers

- **Compilers** take the entire program from start to finish and convert it entirely into object code.
- Different systems will require different object codes and therefore different compilers.
- This initial compilation process can be quite slow, however once done, the object code can now be distributed to any one on the compatible system.
- The original source code is not needed on the target machine.
- It is much faster when running (after being compiled) than an interpreter.
- If it needs to be updated, the source code has to be changed and then compiled.

Bytecode

- Many languages are not only compiled or interpreted, there are various possibilities in between.
  - Most interpreted languages such as Java and Python use an intermediate representation.
  - The code is compiled resulting in bytecode which is then executed by a bytecode interpreter.
  - An advantage of the bytecode is that you achieve platform independence, any computer can run a Java program as long as it has a Java Virtual Machine (JVM).
  - The JVM understands bytecode and converts it into machine code.

Closed Source Software

- Software which is owned by someone or an organisation.
- The software is sold commercially in the form of a license.
- The software is precompiled to machine code.
- This means the user can simply run the machine code on their computer, they have no need for, and no access to the source code.
- The source code is kept by the creators.

Linkers, Loaders and Libraries

- When writing computer code, there is no point in re-inventing the wheel. A lot of tasks have already been written that could be used in the code. When this is the case, they can, and should be recycled by the programmer. An easy way to do this is through the use of libraries.

Libraries

- Libraries are ready-compiled and tested programs that can be run when needed.
  - They are typically grouped together into software libraries.
  - Most programming languages have extensive libraries of pre-built functions.
  - For example, the *math* library in Python provides common solutions to many everyday problems encountered by programmers who are dealing with numbers.
- Someone programming in the Windows Operating System can call Dynamic Link Libraries (DLL).
- These libraries contain subroutines written to carry out common tasks on the Windows OS e.g. Save As, where the user needs to save their work as a file.
- All the programmer needs to do is call the appropriate DLL sub-routine with the correct parameters, and the Save As dialogue box will appear.

Benefits of Use

- Quick and easy to use and hook into your own code.
- Pre-tested, so you can be relatively sure they are already free from errors.
- Pre-compiled, so they are typically optimised to run quickly.

Drawbacks of Use

- Adding functionality or making specific tweaks can be difficult - or impossible.
- Sometimes you are “black-boxed” from actual implementation.
- You have to trust that the developers will continue to maintain the library.

Linkers

- The linker is responsible for putting the appropriate machine addresses in all the external call and return instructions so all modules and external library routines are linked together correctly.
- It also links any separately compiled subroutines into the object code.
- The linker can use two methods to pull in the libraries it needs:

Static Linking

- All the required code from the libraries is included directly in the finished machine code - this can result in large executable program files.

Dynamic Linking

- Compiled versions of the required libraries are stored on the host computer.
- The operating system links the required code from the library as the program is running.
- While this cuts down on the size of the compiled machine code, if the dynamic libraries change, the program may stop because it tries to call a subroutine in a wrong way.

Loader

- The loader is the part of the operating system that loads the executable program file (machine code) into memory, ready to be run.
- When using dynamic linking, it will also be responsible for loading the required libraries into memory.

Stages of Compilation

- Programmers write source code, which is close to English.
- A CPU needs to turn source code into machine code through the process of compilation.
- Compilers run through a series of parses, with each parse of the code, a compiler performs different actions on the source code.
- Each parse is designed to carry out a set task.
- There are four stages of compilation.

Stage 1: Lexical Analysis

- The lexer starts by converting lexemes in the source code into a series of tokens.
- As the lexer reads the source code, it scans the code letter by letter.
- When it encounters a white space, operator symbol or special symbol, it decides that a word (lexeme) is complete.
- It then checks if the lexeme is valid using a predefined set of rules that allow every lexeme to be identified as a valid token.
- Keywords, constants, identifiers, strings, numbers, operators and punctuation symbols are all considered **tokens**:

Example Tokens

Aa Token Class	Example
Identifier	Any function or variable name
Keyword	As If Else End Function Return
Separator	() &
Operator	+-*/%^ DIV MOD < <= > >=
Literal	Hello World
Number	-4 0 3.4
Quote	“”
Bool	True False
Datatype	Integer Decimal String Boolean

- Once the token streams from the lexemes in the source code have been created, it is time to input those tokens into a symbol table.

Stage 2: Syntax Analysis

- Syntax analysis is the second phase of the compiler design process and comes after lexical analysis.
- It receives its inputs in the form of tokens from lexical analysers.
- It analyses the syntactical structure of the input, checking if it is in the correct syntax of the programming language it has been written in.
- It does this by analysing the token stream against production rules to detect any errors in the code.
- Doing this accomplishes two tasks:
    - Checking for errors and reporting them.
    - Building an abstract syntax tree (parse tree).
- The lexer cannot tell us if tokens are valid within the grammar (syntax) of the language - but the syntax analyser can.
- Code will pass the syntax check if it strictly follows the rules of the language.
- If the check fails, the syntax analyser can report the failure to the user, letting them know the exact line and location of the error and even suggesting possible corrections.
- Aside from producing errors if the program fails, the main output of this phase is to create an abstract syntax tree (parse tree).
- The abstract syntax tree is created from the input token stream.
- As it is created, it is strictly following the syntax diagrams to check everything is correct.
- When an identifier is added to the abstract syntax tree, the symbol table is checked to see if it exists.
- The information from the abstract syntax tree can be used to update the data type of identifiers.

Stage 3: Code Generation & Stage 4: Code Optimisation

- In these final phases, the machine code is generated.
- Code optimisation attempts to reduced the execution time of the program by:
    - Spotting redundant instructions and producing object code that achieves the same effect as the source program - but not necessarily by the same means.
    - Removing subroutines that are never called.
    - Removing variables and constants that are never referenced.
- Code optimization can considerably increase compilation time for a program.

- At source-code level, the user can write anything they like.
- At a compilation level, the compiler can search for instructions redundant in nature - e.g. multiple loading and storing of instructions may carry the same meaning even if some of them are removed.
- The compiler could delete the first instruction and rewrite the line of assembly code as shown here.
- Unreachable code is part of the program code that is never accessed because of programming constructs.
- Programmers may accidentally write a piece of code that can never be reached.
- There are instances in code where the program control jumps back and forth without performing any significant task.
- These jumps can be removed.

TLDR

Lexical Analysis

- Comments and white space are removed.
- Remaining code turned into a series of tokens.
- Symbol table is created to keep track of variables and subroutines.

Syntax Analysis

- Abstract syntax tree is built from tokens produced in the previous stages.
- Errors generated if any tokens break the rules of the language.

Code Generation

- Abstract code tree converted to object code.
- Object code is the machine code produced before the final step (linker) is run.

Optimisation

- The code is tweaked it will run as quickly as possible and use as little memory as possible.