# Task 1 – Understanding Object-Oriented Programming

A) Explain what the Object-Oriented Programming paradigm is and outline at least three benefits which make it advantageous for software development.

Object-oriented programming is a programming paradigm based on the operation around the objects. These objects can store data in the parameters with closely related code or functionality in the functions (methods). OOP is based on the idea of classes and objects, where they organise the program into basic reusable ‘blueprints’. The use of this paradigm is based on threatening the program as real-world objects, such as the user. It helps to interact with the object and build relationships.

Especially useful for big and complex projects, where collaborative development is a priority, and the project is divided into smaller groups, according to the OOP paradigm, programmers work with these smaller objects. To implement a project with the OOP paradigm, the first is needed to plan the objects and design their relationships, the process known as Data Modelling. Then the Data and Functions are combined to create the Object. Connections between several objects also could be established at this point.

### Advantages of OOP

1. **Modularity for easier troubleshooting**  
   OOP allows developers to work on one piece of software at a time, which leads to better modularity. This approach makes it easier to isolate and fix issues since a problem in one part of the system can often be fixed without affecting other parts. For instance, if an issue arises in a single object (or class), a developer can work on it without needing to comb through the entire codebase, potentially saving time and reducing complexity.
2. **Reuse of code through inheritance**  
   OOP's inheritance allows new objects to take on properties and behaviours of existing objects, making it easy to reuse code. This not only speeds up the development process but also minimizes errors by using already tested and proven code. For example, if multiple classes share common attributes and methods, they can inherit from a single superclass, reducing the need to duplicate code across the software.
3. **Flexibility through polymorphism**  
   Polymorphism enables one interface to be used for a general class of actions. This flexibility allows code to be more easily extended and maintained. For example, a single function can work on objects of different classes, and specific implementations can be chosen at runtime. This means that changes in one part of a system can be made with minimal to no impact on other parts, enhancing the system's ability to evolve.
4. **Improved productivity and quality of code**  
   The structured nature of OOP facilitates clearer programming logic and shorter code. The reuse of previously written code can significantly accelerate the development process. Moreover, because OOP paradigms encourage more thorough planning and design in the early stages of software development, the resulting code is often of higher quality and more closely aligned with business objectives.
5. **Better data security and handling**  
   Encapsulation provides a way to restrict access to object components, thus protecting the object's integrity by preventing unintended or harmful modifications. This means that the internal state of an object can only be changed by an object's methods, which enforces a controlled access mechanism to the object's data. This level of data protection is particularly valuable in large systems or projects where maintaining data integrity is crucial.

B) Explain the concepts of abstraction and encapsulation in Object-Oriented Programming, and outline how they contribute to creating well-structured and maintainable software systems.

### 1. Encapsulation

Encapsulation is one of the core principles of OOP and it is a process of bundling data (parameters) and functions (methods or procedures) into a single object. Encapsulation intends to hide the internal state of an object from the outside. Access to the ‘private’ data parameters is implemented through the ‘public’ methods, which do not allow for changing the parameters directly, but only through these public functions with proper control and validation. Encapsulation highly enhances security by preventing accidental or unauthorized modifications to the data.

*Real-world example*

Imagine we're creating a software system for a training centre that allows us to manage a list of students in various courses. This system should be able to add new students, update student details, and remove students from the course. We will use encapsulation to achieve these functionalities securely and efficiently.

* The Student Class represents each student with private attributes such as student\_id, first\_name, last\_name, age, fees\_due and course. It also includes methods to get and set these attributes, ensuring that any changes to the student's details are controlled and validated.
* The StudentManager Class this class manages all students within a course. It includes a private list that holds the students. The StudentManager provides public methods to interact with this list.

### 2. Abstraction

Abstraction is the second fundamental principle of OOP that focuses on exposing only the key features of an object while hiding unnecessary details and background functionality. It allows us to handle complexity by presenting at a more simple level rather than overloading with the details. In simple terms, abstraction is about identifying what an object does rather than how it achieves what it does. The key benefits of abstraction are simplicity, focus on what matters and flexibility.

*Real-world example*

Let's continue with the scenario of creating a software system for a training centre, focusing on the management of a list of students in various courses.

* The StudentManager class **simplifies** the complexity of managing students by providing a high-level interface. Users of the StudentManager do not need to know how students are stored or how the operations are implemented.
* By abstracting the details of Student Management, the system allows users to **focus** on what actions can be performed (add, edit, remove) rather than how these actions are executed. This separation of concerns makes the system easier to understand and use.
* The abstract of the StudentManager allows the underlying implementation of how students are managed to change (e.g., moving from a list to a database) without affecting the rest of the system. This **flexibility** is a direct benefit of abstraction.

### 3. Polymorphism

Polymorphism, a third core concept in OOP, enables objects of different classes to be treated as objects of a common superclass. It allows methods to do different things based on the object it is acting upon, even though they share the same name. This is achieved through two main types: static polymorphism and dynamic polymorphism. Static polymorphism is achieved through method overloading, while dynamic polymorphism is achieved through method overriding.

*Real-world example*

Continuing with our training centre software system, let’s imagine that our Student class has the get\_course() method and this method doesn’t return all information about the students, only the course itself, which could be not enough for the StudentManager that wants the course name, first\_name and last\_name. To achieve this we can implement dynamic polymorphism and override the get\_course() method in StudentManager class to return this additional information.

### 4. Inheritance

Inheritance is the final main concept in OOP that allows a class to inherit properties and methods from another class. The class that inherits is known as the subclass (or derived class), and the class from which it inherits is known as the superclass (or base class). Inheritance facilitates code reusability, enabling new objects to take on existing properties and behaviours of other objects while introducing their unique features.

*Real-world example*

In our example of a training centre software system, imagine we need to expand our system to include apprenticeship members alongside third-level students. Both third-level and apprentices share common attributes such as name, ID, and contact details, yet they also have unique attributes and functionalities.

* Students Class acts as a superclass for both third-level students and staff, containing shared attributes and methods like getName, getID, and getContactDetails.
* Third-level class and Apprentices Class. Both classes inherit from the Students' class but also implement their unique properties and methods. The Student class might include attributes and methods related to courses and fees, whereas the Apprentices class could include attributes and methods relevant to their responsibilities, like employer.

# Task 2 – Designing an Object-Oriented Programming solution

Create a UML diagram for the design of a basic bank account. The design should account for two distinct types of accounts, savings accounts and current accounts. Current accounts will have set transaction fees which will apply to all withdrawals. Savings accounts will have a set interest rate which will be used to calculate interest for the account (balance \* interest rate). Below is a breakdown of the attributes and functionality which should be represented by your class design.

A screenshot of a computer screen

Description automatically generated

### Classes Description and Relationships

* **Bank\_Accounts** - is the base class for bank accounts. It includes common properties like account\_number, first\_name, last\_name, and methods for depositing, withdrawing, and checking the balance.
  + Attributes:
    - **+account\_number: str** - Public attribute indicating the unique account number.
    - **+first\_name: str** - Public attribute indicating the account holder's first name.
    - **+last\_name: str** - Public attribute indicating the account holder's last name.
    - **-\_balance: float** - Private attribute holding the account balance, to warn from the outside usage, should be used directly, only through the functionality methods down below.
  + Methods:
    - **+deposit(amount: float): bool** - Public method to deposit money into the account. Returns True if successful.
    - **+withdraw(amount: float): bool** - Public method to withdraw money from the account. Returns True if successful.
    - **+get\_balance(): float** - Public method to get the current balance.
* **Current\_Account** - This class inherits from Accounts. It overrides the withdrawal method to include a transaction fee. This is a clear use of inheritance, where Current\_Account extends Bank\_Accounts to add specific behaviour.
  + Unique Attribute:
    - **+transaction\_fee: float** - Public class attribute for the transaction fee.
  + Overridden Method:
    - **+withdraw(amount: float): bool** - Overridden public method to withdraw money, including the transaction fee.
* **Savings\_Account** - This class also inherits from Bank\_Accounts and introduces a new method calculate\_interest to calculate interest earnings on the savings account balance. Again, this demonstrates inheritance, as Savings\_Account shares common features with Bank\_Accounts but adds unique functionality.
  + Unique Attribute:
    - **+transaction\_fee: float** - Public class attribute for the transaction fee.
  + Unique Method:
    - **+withdraw(amount: float): bool** - Overridden public method to withdraw money that includes the transaction fee to the calculation.

# Task 3 – Implementing an Object-Oriented design

Task 3 is implemented in the ‘ONAssignment\_OOP.py’ file, which holds all Banking Application functionality. Please consider, that they implemented two interfaces, but only the CLI interface is finished when GUI is under development and should reflect the application implementation.

# Task 4 – Understanding Test-Driven Development

A) Briefly explain the concept of test-driven development and outline the benefits it can have on the software development process.

Test-Driven Development is a software development approach that emphasizes the creation of tests before writing the actual code. This methodology reverses the traditional development process on its head, requiring developers to think about their code's functionality and edge cases upfront. By focusing on tests first, TDD aims to ensure that the developed software meets all specified requirements from the outset, leading to higher quality and more reliable code.

### TDD Life Cycle

* Writing a Failing Test phase is called the **Red Phase**. The cycle begins by writing a test that defines a desired improvement or new function. This test will naturally fail when first run because the feature it tests does not yet exist. This red phase validates that the test is correctly detecting the absence of the feature.
* Make the Test Pass phase is called the **Green Phase**. It’s the next step to write the minimum amount of code necessary to make the test pass. This phase focuses on functionality rather than optimization. The goal is to quickly get to a green passing state, confirming that the code meets the requirements defined by the test.
* And last is the **Reactor Phase**. Once the test passes, the code is then refactored to improve its structure, readability, and performance while at the same time ensuring that it still passes all tests. This phase helps maintain code quality over time without altering its external behaviour.

### Three key benefits of Test Driven Development

TDD brings several key benefits to the software development process.

1. Firstly, it greatly **improves code quality**. By writing tests first, developers are forced to consider the functionality of their code from the beginning, leading to fewer bugs and errors. This approach also ensures that every piece of code has a corresponding test, which can significantly enhance the robustness of the software.
2. Another benefit is **enhanced documentation**. The tests themselves serve as a form of live documentation. Since they describe what the code is supposed to do, other developers can quickly understand the intended functionality without having to dig deeply into the implementation details. This aspect is particularly useful in large teams or when handing over projects.
3. Lastly, TDD facilitates **better design and architecture**. Writing tests first encourages developers to think about the design of their system upfront. This leads to more modular and flexible code, as developers are naturally inclined to write smaller, more focused functions that are easier to test. This early focus on design can reduce the need for costly refactoring later in the development process.

B) Based on the design from Task 2 and subsequent code created in Task 3, explain how unit testing could be implemented and create three unit tests to demonstrate their functionality.

To effectively demonstrate the functionality of the Banking Application, we'll write three unit tests covering key features of the application: depositing into an account, withdrawing from an account, and calculating interest for a savings account. Each test will follow the structured approach of the Tested Feature, Code Snippet, Code Explanation, Unit of Code Description, Inputs, Expected Results and Actual Results.

### Test 1: Deposit Functionality

1. **Tested Feature**

The main functionality of our Bank Application is a Deposit to a Bank Account. This function is written in the parent ‘Bank\_Accounts’ class. There is also a UI element for handling the deposit interaction, but we will be focusing on the main functionality.

1. **Code Snippet**

    def deposit(self, amount: float) -> bool:

        if amount > 0:

            self.\_balance += amount

            return True

        return False

1. **Code Explanation**  
   In this deposit function, we perform several checks. First, we ensure that the amount we try to deposit is greater than 0, in this case, the goal is to avoid potential negative inputs that could erroneously deduct from the account balance. Second, we return True if the operation was successful and False if wasn’t. This return functionality is needed for the UI to react properly to some mistakes.
2. **Unit of Code Description**

The ‘deposit’ method in the ‘Bank\_Accounts’ class should correctly add the specified amount to the account's balance.

1. **Inputs**

An instance of ‘Bank\_Accounts’ for John Doe’s Current Account with an initial balance of €0.00, and a deposit amount of €500.00.

1. **Expected Results**

The balance of the account should be updated to €500.00 after the deposit.

1. **Actual Results**  
   A screenshot of a computer screen

   Description automatically generated

### Test 2: Withdraw Functionality

1. **Tested Feature**

The second main functionality of our Bank Application is a Withdraw from an Account, in our case we will use Current Account. According to our requirements, Current Accounts have set transaction fees which will apply to all withdrawals of €1.50. This function is written in the parent ‘Bank\_Accounts’ class but also overwritten in the Current Account class as it contains transaction fees, that uniquely apply to this type of account. There is also a UI element for handling the deposit interaction, but we will be focusing on the main functionality.

1. **Code Snippet**

    def withdraw(self, amount: float) -> bool:

        total\_amount = amount + Current\_Account.transaction\_fee

        if self.get\_balance() >= total\_amount:

            if super().withdraw(total\_amount):

                return True

        return False

1. **Code Explanation**  
   In this withdraw function, we also perform several checks. First, we ensure that the account has sufficient funds to execute the operation, in this case, we want to avoid the situation when the balance will become negative if we don’t calculate the transaction fee properly. Second, we return True if the operation was successful and False if wasn’t. This return functionality is needed for the UI to react properly to some mistakes.
2. **Unit of Code Description**

The withdraw method in the ‘Current\_Account’ class should correctly subtract the specified amount plus a transaction fee from the account's balance.

1. **Inputs**

An instance of the ‘Current\_Account’ on the name of ‘John Doe’ with an initial balance of €1000.00, and a withdrawal amount of €200.00. We will pre-input this scenario through the Command Line User Interface.

1. **Expected Results**

The balance of the account should be updated to €798.50 after the withdrawal, considering a €1.50 transaction fee.

1. **Actual Results**  
   A screenshot of a computer screen

   Description automatically generated

### Test 3: Interest Calculation

1. **Tested Feature**

One of the required functionality is the ability to calculate interest for the Savings Accounts. Savings Accounts have a set interest rate which is used to calculate interest for an account. The calculation formula is balance multiplying by interest rate of 2.5%.

1. **Code Snippet**

    def calculate\_interest(self) -> float:

        return self.get\_balance() \* Savings\_Account.interest\_rate

1. **Code Explanation**

This unique function is special to the ‘Savings Account’ class. We also store the ‘interest\_rate’ value inside its class to have the ability to set it in our program later in case of policy changes. We also call get\_balance(), the function from our parent class, to receive balance from the private attribute, instead of referring to self.\_balance itself, as it’s not safe to do so.

1. **Unit of Code Description**

The ‘calculate\_interest’ method inside the ‘Savings\_Account’ class should calculate the interest earned on the account's balance.

1. **Inputs**

An instance of ‘Savings\_Account’ for ‘John Doe’ with a balance of €1000.00. We will pre-input this scenario through the Command Line User Interface.

1. **Expected Results**

The interest calculated based on an interest rate of 2.5% should be €25.00.

1. **Actual Results**A white screen with black text

   Description automatically generated