**What is OO?**

A Paradigm or way of thinking / viewing / structuring for software

**Define OOP?**

* A paradigm is very centralized around object concept where class collect data then enforcement
* Figure out object + function + interactions

**Why OOP?**

* **Real-world analogy**: Makes it easier to model real-world systems.
* **Complexity management**: Helps handle large, complex systems by organizing them into smaller, manageable parts (classes).
* **Maintainability**: Promotes easier maintenance, debugging, and testing.
* **Code reuse**: Encourages code reuse and reduces redundancy.
* **Extensibility**: Easily extendable and adaptable to new requirements.
* **Avoiding software crisis**: Helps avoid over-budgeting, overtime, and bugs in large-scale systems.

**What is Procedural Programming?**

* Paradigm focus is on creating a sequence of **steps**
* Control flow is based on **sequential steps** (like loops, conditionals) that modify the state as the program runs.
* Data is often **mutable** and passed to functions where it can be modified.
* The program is usually a collection of tasks
* Functions are used to encapsulate **procedures** or steps that change the state of the system
* Suitable for applications with well-defined, sequential operations

**What is Functional Programming?**

* Paradigm focus is on **pure functions** and **immutable data**.
* Control flow is based on **function calls** and recursion (iteration is usually done via recursion, not loops).
* Data is **immutable**: once created, data cannot be changed. Instead, new data is returned from functions.
* Functions are first-class citizens (assign to variable – pass to function parameter- return function from function)
* Functions given the same inputs, they always return the same outputs and do not modify any external state.
* Suitable for more complex programs where modularity, reusability, and correctness are priorities

**Flaws of Functional Programming?**

* Difficult to handle systems that require frequent state changes
* Recursive functions can cause performance hits or stack overflow if not optimized

**Flaws of Procedural Programming?**

* No owner for data so it can be found everywhere
* Data integrity issue and Duplication
* Missing Data ex: customer address is missing in order (on validation)
* Inconsistent Data (string CutomerName= "Jhon" in place and "Mike"in another part of the program)
* Hard debugging if data corrupted or to update it

**What is the class?**

* **Blueprint** or template or that describes the state and behavior of objects
* Pattern or container or of a given type of copies objects

**What`s class element?**

Declaration: define class name

Body: curly brackets container

Fields: data members defined

Methods: member functions perform operations

Properties: characteristics assigned by access modifiers

**How it works?**

Group all relevant attributes in ONE unit

Provides internal operations on these data using methods

Bind variables **attributes** & operations **methods**

**What is object?**

Tangible entity that represents a specific instance of a given class **attributes**

**Attributes** State: define or describe object in a specific moment as data member

**Methods** Behavior: the specific action that can be done by object which make changes

**What is Encapsulation?**

Group all the data members & member functions within one single unit where can access them using

Setter & Getter

**Why we use encapsulation?**

**Data Hiding**: restricts direct access to some of an object's components only allows changes through well-defined (getters and setters)

**Code Organization**: grouping related data and methods together, reducing complexity and making software manageable

**? Define Accessors & Metuters**

Accessors: is a function used to read / access data member

Mutators: is a function used to change/mutate data member

**“**Don`t ask for the information you need to do the work, ask the object for the information that has to do the work” **provide more functions than use set & get**

**Different Bet~ Data Hiding & Encapsulation?**

Hiding data/functions from outside code using Access modifiers

1. **Public**:
   * The member is accessible from anywhere, both inside and outside the class.
   * No restrictions on access.
2. **Private**:
   * The member is accessible only within the class where it is declared.
   * Cannot be accessed directly from outside the class.
3. **Protected**:
   * The member is accessible within the class and by subclasses (derived classes) but not from outside the class or package.
   * Allows inheritance to access the members.
4. **Default (Package-Private)**:
   * In languages like Java, if no access modifier is specified, the member is accessible within the same package but not outside it.
   * This is also called package-private in some languages.
5. **Internal (C# specific)**:
   * The member is accessible within the same assembly or project but not outside it.
   * Used to restrict access within a specific project.
6. **Protected Internal (C# specific)**:
   * Combines protected and internal. The member is accessible within the same assembly or by derived classes.

**Why we use Data Hiding?**

Reduce outsider’s dependency (coupling) to whole data/functions

Hide complexity & object state **changes ex: Google search engine**

**Different between public variable and private variable uses setter & getter?**

**Accessibility**

**Public** is directly accessed from anywhere (no control or validation) its value can affect behavior of code throughout a program

**Private** Indirect access, allowing you to control how the data is accessed and modified using validations (more secure and flexible)

**Use Public Variables** when:

* The value is constant or immutable.
* You want the variable to be directly accessible and you don’t need validation or protection.
* The class is simple, and you're okay with exposing all its fields.

**Ex:** DTOs (Data Transfer Objects), simple containers of data

**Use Private Variables with Setters/Getters** when:

* You need to protect the data or ensure it remains valid.
* You need flexibility to change the internal implementation without breaking external code.
* You want to encapsulate logic for validation, security, or other behaviors.

**Ex**: Suppose you have a Person class with an age variable. You might want to ensure that the age is always positive and never negative. If the age was public, someone could directly set it to a negative number, which might not make sense in your program.

**Abstraction**

Is about hide **more** unwanted details & showing most essential in a context

**How** hide as possible you can implementation and data

show (What) user will use & context identify what`s needed and what`s not

**Context**

Specific environment or circumstances under which a piece of code is executed

Influenced by factor (variables - state - scope - calling) ex: employee (Google – Toyota)

**Why we need Abstraction?**

**Coupling:** class directly uses another class's methods or data without any abstraction (tight)

**Risk of Errors:** Exposing internal details without abstraction leads to side effects & require changes in the dependent class

**Flexibility and Scalability:** Adding new features or extending functionality becomes harder because changes might ripple through the entire codebase.

**Abstract class & Concrete class & Interface?**

|  |  |  |  |
| --- | --- | --- | --- |
| **Interface** | **Abstract Class** | **Concrete class** | **Features** |
| Can`t be instantiated | Can`t be instantiated | Can be instantiated directly | **Instantiation** |
| Can be implemented by multiple classes (multiple inheritance) | Supports inheritance (single inheritance) | Can be inherited (single inheritance) | **Inheritance** |
| Contains only abstract method declarations (or default methods in some languages) | Can contain both abstract and concrete methods | Contains only concrete methods | **Methods** |
| Complete abstraction (no method implementations) | Partial abstraction (abstract methods and concrete methods) | No abstraction | **Abstraction** |
| Used to define a contract that implementing classes must follow | Used as a base class for other classes | Direct use in code | **Use case** |
| Can`t have a constructor | Can have a constructor | Can have a constructor | **Constructor** |

**Virtual & Pure Virtual Method & Friend?**

**Virtual:** A method in a base class with a default implementation that can be overridden or use its implementation in derived classes.

**Pure Virtual:** A method in a base class with a no implementation that must be overridden in derived classes to ensures that the derived classes provide their own behavior

**Friend Function:** Non-member functions useful for cases where you need access to private members but don’t want to make the function a member of the class itself

**Polymorphism**

Allows objects/methods of different types to be treated as if they were of the same type **have many forms same type**

Allow us to write generic code without knowing exact **children**

|  |
| --- |
| **Ex**: void process(Shape shape) -> can be circle - Rectangle - Triangle |

**Overloading**

Two or more methods in the same class have the **same name** but **different parameters (Type + number)**

Allows a class to have multiple methods that perform similar tasks but with different input types or different numbers of arguments

Overloading is resolved during **compile-time** based on the method signature (name + parameter list)

Overloaded methods can have the **same** or **different** return types, but the return type alone cannot be used to distinguish overloaded method

**Overriding**

Occurs when a **subclass** provides its own specific implementation of a method that is already defined in its **superclass**

Change or extend the behavior of the inherited method

Can use the @Override annotation to explicitly indicate that a method is being overridden

Resolved during **runtime**. The method to be executed is determined based on the actual object type, not the reference type

**Virtual Table**

A mechanism used by **OOP** to implement **runtime polymorphism** and **method overriding**

Virtual table allows dynamic dispatch or late binding

Method that should be executed is determined at **runtime**, based on the actual object type, rather than the reference type.

**How Virtual Table Works:**

* An array of pointers to methods that marked as virtual
* Each class with virtual methods (methods that can be overridden in derived classes) has its own vtable.
* Every object of a class that has virtual functions contains a pointer to the vtable (often called a vptr).
* The vptr is set up when an object is created, and it points to the vtable corresponding to the class of the object.
* When a method is called on an object, the program uses the vptr to look up the correct function in the vtable. This allows the program to call the overridden method of the actual class of the object, not the class of the reference

**Inheritance**

Represents an "is-a" relationship between classes (sub class - superclass)

Allows a class to **copy** properties, methods, and other characteristics from another class

**Base class**: A class that is inherited by another class and it is starting point for other classes. It can be called as general class.

**Super class**: A class that is inherited by another class can be called parent class

**Derived class**: A class that inherits from a base or super class

**Inheritance Abuse**

**Overgeneralization**: this occurs when a **superclass** is created too broadly, meaning it tries to capture too many behaviors and properties from a variety of subclasses, which leads to a **lack of specificity**

**Ex**: Superclass **Vehicle** that includes all possible vehicle-related behavior, such as drive (), sail (), and fly (), even though not all subclasses (e.g., Car and Boat) would use all these behaviors.

**Under generalization superclass** fails to capture the **commonalities** between subclasses that could be shared that leads to subclasses duplicate similar behavior

**EX:** Car and Truck share common properties like engine () and wheels (), but those properties are not extracted into a Vehicle superclass.

**Diamond Problem**: a class inherits from **two or more classes** that share a **common ancestor**, leading to ambiguity in resolving inherited methods or attributes.

**Types of Inheritance**

**Single**: A subclass inherits from only one superclass

**Multiple**: A subclass inherits from multiple super classes **not supported in some languages**

**Multilevel**: A subclass inherits from another class, which itself inherits from a superclass

**Hierarchical**: Multiple subclasses inherit from a common superclass

**Hybrid**: A combination of multiple and hierarchical inheritance

Constructor

Deconstructor

Keywords (static – let – var – val – final – const –sealed – factory )

Binding -> late – early

Copy by value – reference

Static & dynamic memory

Compile & run Time

SOLID

Design patterns

Creational: factory –singleton – abstract – builder – prototype

Structural: adaptor – decorator –façade – composite – proxy

Behavioral: chain - command – observer – state – strategy