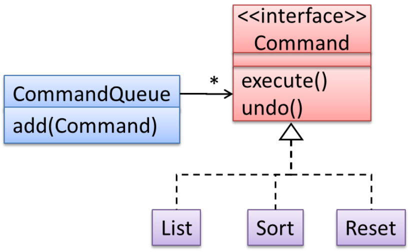
Principle

Single responsibility principle (SRP)

- A class should have one, and only one, reason to change. **[about classes not methods]**
- If a class has only one responsibility, it needs to change only when there is a change to that responsibility

Open-Closed Principle: A module should be **open for extension but closed for modification**.

- modules should be written so that they can be **extended, without requiring them to be modified**
- aims to make a code entity **easy to adapt and reuse** without needing to modify the code entity itself.
- often requires separating the specification (i.e. interface) of a module from its implementation.



Liskov substitution principle (LSP)

- a subclass should not be **more restrictive** than the behavior specified by the superclass.
- If class B is substitutable for parent class A → should pass all test cases of parent class A. Otherwise, not substitutable and violate LSP.

SOLID Principles [five OOP principles]

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Separation of concerns principle (SoC)

- To achieve **better modularity**, separate the code into distinct sections, such that each section addresses a separate concern.
- **reduces functional overlaps** among code sections
- **limits the ripple effect** when changes are introduced to a specific part of the system.
- can be applied at the class level, as well as at higher levels
- lead to **higher cohesion** and **lower coupling**.

Law of Demeter (LoD)

- An object should have **limited knowledge** of another object.
- An object should **only** interact with objects that are **closely related** to it.
- Also called **"Don't talk to strangers"** and "**Principle of least knowledge**"
- aims to **prevent objects from navigating** the internal structures of other objects.

Example: a method **m** of an object **O** should invoke only the methods of the following kinds of objects:

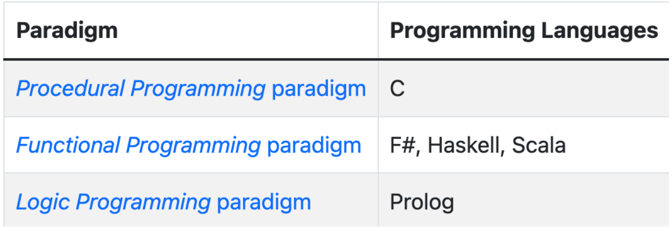
- The object O itself
- Objects passed as **parameters of m**
- Objects **created/instantiated in m** (directly or indirectly)
- Objects from the **direct association** of O

OOP

Object-Oriented Programming (OOP)

- is a **programming paradigm**
- views the world as a **network of interacting objects**
- **OOP solutions** try to create a similar object network inside the computer’s memory
- **does not** demand that the virtual world object network follow the real world exactly

Programming paradigm guides programmers to **analyze** programming problems, and **structure** programming solutions, in a specific way.



Paradigm	Programming Languages
Procedural Programming paradigm	C
Functional Programming paradigm	F#, Haskell, Scala
Logic Programming paradigm	Prolog

Java

- is primarily an OOP language
- supports **limited forms** of functional programming
- can be used to (not recommended) write procedural code

JavaScript and **Python support**

- functional
- procedural
- OOP

An **Object** in OOP

- has both **state** (data) and **behavio**r (operations on data), similar to objects in the real world
- has an **interface** and an **implementation**
- interact by sending **messages**
- is an **abstraction mechanism** [allows us to abstract away the lower level details and work with bigger granularity entities]
- is an **encapsulation** of some data and related behavior in terms of **packaging aspect** and **information hiding aspect**.

The packaging aspect: An object packages data and related behavior together into **one self-contained** unit.

The information hiding aspect: The data in an object is **hidden from the outside** world and are only accessible using the object's interface.

Class: contains instructions for creating a specific kind of objects

Class-level members: Class-level attributes and methods

Enumeration

- is a fixed set of values
- can be considered as a **data type**.
- useful when using a regular data type such as int or String would allow **invalid values** to be assigned to a variable.

Associations: connections between objects

- **main connections** among the classes in a class diagram.
- can **change** over time
- can be **generalized** as associations between the corresponding classes
- implemented by using **instance level variables**
- can be shown as an **attribute** instead of a line.
  name: type [multiplicity] = default value
  Show each association as **either an attribute or a line** but **not both**. A line is preferred as it is easier to spot.

Association class

- represents **additional information** about an association
- is a **normal class** but plays a special role from a design point of view.
- can be implemented as a **normal class** with variables to represent the endpoint of the association it represents.

Navigability tells us if an object taking part in association knows about the other.

- unidirectional or bidirectional
- The **arrowhead** (not the entire arrow) denotes the navigability. The line denotes the association
- is an **extra annotation** added to an association line

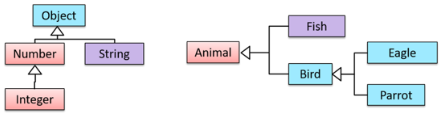
Multiplicity: is the aspect of an OOP solution that dictates how many objects take part in each association

- A normal instance-level variable gives us a 0..1 multiplicity (also called optional associations)
- A variable can be used to implement a 1 multiplicity too (also called compulsory associations).

Inheritance allows you to define a new class based on an existing class.

- A superclass is said to be more general than the subclass.
- implies the derived class can be considered as a subtype of the base class, resulting in an **is a** relationship.
- Inheritance relationships through a chain of classes can

result in **inheritance hierarchies** (aka **inheritance trees**).



- UML notes can augment UML diagrams with additional information. These notes can be shown connected to a particular element in the diagram or can be shown without a connection.
- **Multiple Inheritance** is when a class inherits directly from multiple classes. Multiple inheritance among classes is allowed in some languages (e.g., Python, C++) but **not in other languages (e.g., Java, C#)**.

Method overriding

- same name
- same type signature
- same (or a subtype of the) return type
- **overridden methods** are resolved using **dynamic binding**, and therefore resolves to the implementation in the actual type of the object.

Method overloading: indicate that multiple operations do similar things but take different parameters.

- same method name
- different method signatures
- possibly different return types.
- **overloaded methods** are resolved using **static binding**

Type signature: type sequence of the parameters. The return type and parameter names are **not** part of the type signature. However, the parameter **order is significant**.

Substitutability: Ability to substitute a **child class object** where a parent class object is expected.

- instance of a subclass is an instance of the superclass
- inheritance allows substitutability

Dynamic binding (Late binding): method calls in code are resolved at **runtime**, rather than at compile time.

Static binding (Early binding): method call is resolved at **compile time.**

Polymorphism: allows you to write code targeting superclass objects, use that code on subclass objects, and achieve possibly different results based on the actual class of the object. **Achieve polymorphism:**

- substitutability
- operation overriding
- dynamic binding

Composition is an association that represents a strong whole-part relationship.

- When the whole is destroyed, parts are destroyed too
- **Cannot be cyclical links**.
- Whether a relationship is a composition can depend on the context. In other words, two objects may have different relationship in different context.
- A common use of composition is when parts of a big class are carved out as smaller classes
- Cascading deletion alone is **not sufficient** for composition.
- Identifying and keeping track of composition relationships in the design has benefits
- Composition is **implemented** using a normal variable.

Aggregation: a **container-contained** relationship.

(a weaker relationship than composition)

- Containee object can exist even after the container object is deleted.
- Martin Fowler’s famous book UML Distilled **advocates omitting** the aggregation symbol altogether because using it adds more **confusion** than clarity.

Dependency is a need for one class to depend on another without having a direct association in the same direction.

- We are specifically focusing on non-obvious dependencies here
- An association is a relationship resulting from one object keeping a reference to another object.
- we need not show that as a dependency arrow in the class diagram if the association is already indicated in the diagram. [does not add any value to the diagram]
- Use a dependency arrow to indicate a dependency only if that dependency is not already captured by the diagram in another way

Abstract Class: You can declare a class as abstract when a class is merely a representation of commonalities among its subclasses.

- A class that has an abstract method becomes an abstract class

MISC

What is the difference between a Class, an Abstract Class, and an Interface?

Interface: behavior specification with no implementation.

Class: behavior specification + implementation.

Abstract Class: behavior specification + a possibly incomplete implementation.

Model and Diagrams 1

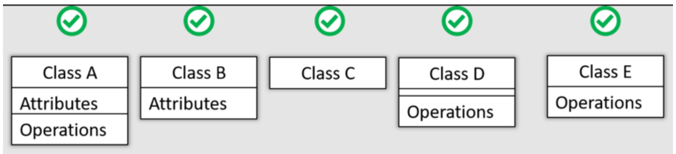
Model

- provides a simpler view of a complex entity
- captures only a selected aspect
- are **abstractions**.
- Multiple models of the same entity may be needed to capture it fully.
-

Usage

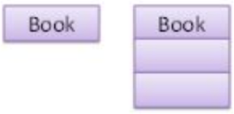
- To analyze a complex entity in software development.
- To communicate information among stakeholders. [As a visual aid in discussions and documentation]
- As a blueprint for creating software

Class Diagram: Describe the **structure** (**not behavior**)



Example: These two are not same

- the first one omits the attributes
- the second one has empty attributes and operations



Visibility: (Not Accessibility)

- no default visibility in UML.
- not show the visibility means unspecified

Model-driven development (MDD):

- Model-driven engineering
- an approach to software development that strives to exploit models as blueprints

Unified Modeling Language (UML): is a graphical notation to describe various aspects of a software system.

Object structures

- can change over time based on a set of rules **set by the designer** of that software. [Not Random]
- Rules that object structures need to follow can be illustrated as a **class structure**

Domain modeling is modeling the problem domain. Useful in understanding the problem domain. Can be done using,

- **a domain-specific modeling notation** if such a notation exists (e.g., a modeling notation specific to the banking domain might have elements to represent loans, accounts, transactions etc.)
- **a general purpose modeling notation**, such as **UML**
- other **general purpose notations** (e.g., **organization chart** to model the employee hierarchy of a company).

Conceptual Class Diagrams(CCDs): UML model captures class structures in the problem domain.

- a **lighter version** of class diagrams
- sometimes also called **OO domain models** (OODMs) [somewhat misleading]
  CCDs are only **one type of domain models** that can model an OOP problem domain.

A UML **sequence diagram** captures the **interactions** between multiple entities for a given scenario.