Object-Oriented Programming

Principles, Pillars, and the Bridge from UML to Code

Abstraction • Encapsulation • Inheritance • Polymorphism

Structured Reference Notes

Table of Contents

Session 1: Introduction to Object-Oriented Programming

1.1 What is Object-Oriented Programming?

Object-oriented programming (OOP) is a programming paradigm that organises software around objects — self-contained units that bundle data (attributes) and behaviour (methods) into a single entity. Rather than structuring a program as a sequence of instructions that operate on separate data structures (as in procedural programming), OOP models the problem domain as a collection of interacting objects, each responsible for managing its own state and exposing a well-defined interface to the rest of the system.

This paradigm mirrors the way humans naturally conceptualise the world. We think in terms of things — a car, a bank account, a student — that have properties and can perform actions. OOP translates this intuition into code: a Car object has attributes such as colour, speed, and fuelLevel, and methods such as accelerate(), brake(), and refuel(). By aligning the structure of the code with the structure of the problem, OOP makes software easier to design, understand, maintain, and extend.

The object-oriented paradigm is supported by all major modern programming languages, including Java, C++, Python, C#, and TypeScript. Although the syntax differs between languages, the fundamental concepts — classes, objects, attributes, methods, and the four pillars — remain the same. This resource uses Java as the primary illustration language, with comparative notes for Python and C++ where instructive.

Key Concept — Object-Oriented Programming

OOP is a paradigm that models software as a collection of interacting objects, each encapsulating data and behaviour. It is built on four pillars: abstraction, encapsulation, inheritance, and polymorphism. These principles collectively promote code that is modular, reusable, maintainable, and extensible.

1.2 Procedural vs Object-Oriented Programming

To appreciate the value of OOP, it is helpful to contrast it with procedural programming — the paradigm that preceded it. In procedural programming, a program is structured as a sequence of function calls operating on shared data structures. Functions and data are separate entities: data is defined in one place and passed to functions that manipulate it. While this approach works well for small programs, it becomes increasingly difficult to manage as systems grow in size and complexity.

Aspect	Procedural Programming	Object-Oriented Programming
Structure	Functions operating on separate data	Objects bundling data and behaviour together

Data access	Data is typically global or passed between functions; any function can modify any data	Data is encapsulated within objects; access is controlled through methods
Reuse mechanism	Copy–paste functions or libraries	Inheritance, composition, and interfaces
Modularity	Achieved through functions and modules; no formal encapsulation	Achieved through classes with private data and public interfaces
Extensibility	Adding new types often requires modifying existing functions	New types can be added via inheritance or interface implementation without modifying existing code
Maintainability	Changes to data structures can require changes across many functions	Changes to a class's internals are isolated from the rest of the system

The fundamental shift in OOP is from thinking about what the program does (procedures) to thinking about what things the program manipulates (objects). This shift has proven enormously productive for large-scale software development, where managing complexity, enabling team collaboration, and accommodating evolving requirements are critical concerns.

1.3 Classes and Objects

1.3.1 Classes as Blueprints

A class is a blueprint or template that defines the structure and behaviour common to all objects of a particular type. It specifies which attributes (data fields) every object will have and which methods (operations) every object can perform. A class does not itself hold data — it merely describes the shape that data will take when an object is created from it.

For example, a class BankAccount might define attributes such as accountNumber, ownerName, and balance, and methods such as deposit(amount), withdraw(amount), and getBalance(). The class is the definition; it tells the system what a bank account looks like and what it can do.

1.3.2 Objects as Instances

An object is a concrete instance of a class — a specific entity created from the blueprint at runtime. Each object has its own copy of the attributes declared by the class, initialised to specific values. Multiple objects can be created from the same class, each with different attribute values but sharing the same set of methods.

Continuing the example above, 'Alice’s current account with balance £1,200' and 'Bob’s savings account with balance £5,000' are both objects (instances) of the BankAccount class. They share the same structure and behaviour, but each holds distinct data. Creating an object from a class is called

instantiation, and it is typically done using a constructor — a special method that initialises the object's attributes.

Key Concept — Class vs Object

A class is a blueprint that defines attributes and methods. An object is a concrete instance of a class, created at runtime with its own attribute values. One class can produce many objects. The class defines the type; the object is the thing.

1.4 Attributes and Methods

Attributes (also called fields, properties, or instance variables) represent the data that an object holds. They define the state of an object at any given moment. In Java, attributes are declared within the class body but outside any method. Each attribute has a type (e.g., int, String, double, or another class) and an access modifier (public, private, protected).

Methods (also called operations, functions, or member functions) represent the behaviour of an object — the actions it can perform or the services it provides. Methods operate on the object's attributes and may accept parameters and return values. Well-designed methods encapsulate a single, well-defined responsibility, keeping the class cohesive and easy to understand.

Term	Also Known As	Definition	Example
Attribute	Field, property, instance variable	A named data item held by each object of the class	private double balance;
Method	Operation, function, member function	A named behaviour that objects of the class can perform	public void deposit(double amt)
Constructor	Initialiser	A special method called when an object is created; initialises attributes	public BankAccount(String owner)
Static member	Class variable / class method	Belongs to the class itself rather than to individual objects	private static int count;

1.5 The Four Pillars: An Overview

Object-oriented programming rests on four foundational principles, commonly referred to as the 'four pillars'. Each pillar addresses a different aspect of software quality: abstraction manages complexity, encapsulation protects data integrity, inheritance promotes reuse, and polymorphism

enables flexibility. Together, they form a coherent philosophy for designing software that is robust, adaptable, and maintainable.

Pillar	Core Question	Primary Benefit
Abstraction	What are the essential features of this entity?	Manages complexity by hiding irrelevant detail
Encapsulation	How do we protect this object's internal state?	Ensures data integrity and reduces coupling between components
Inheritance	How do we reuse and specialise existing code?	Promotes code reuse through hierarchical relationships
Polymorphism	How do we write code that works with many types?	Enables flexibility through one interface, many implementations

The following four sessions explore each pillar in comprehensive detail, with definitions, examples, code illustrations, UML representations, and practical guidance for applying each principle effectively.

Session 2: Pillar 1 — Abstraction

2.1 What is Abstraction?

Abstraction is the process of identifying the essential characteristics of an entity while deliberately ignoring details that are irrelevant to the current context. It is the intellectual tool that allows designers and developers to manage the overwhelming complexity of real-world systems by focusing on what matters and hiding what does not.

In everyday life, abstraction is ubiquitous. When you drive a car, you interact with the steering wheel, pedals, and gear stick — you do not need to understand the combustion cycle, fuel injection timing, or the electronic control unit. The car presents an abstraction: a simplified interface that hides the underlying complexity. Software abstraction works in exactly the same way: a class presents a clean interface (public methods) while hiding the intricate internal logic that makes those methods work.

Key Concept — Abstraction

Abstraction is the process of focusing on the essential features of an entity while hiding the background details. In OOP, a class is an abstraction: it captures the essential attributes and behaviours of a real-world entity at an appropriate level of detail for the system being designed.

2.2 Levels of Abstraction

Abstraction operates at multiple levels within a software system. At each level, different details are relevant and different details are hidden. Understanding these levels is critical for producing designs that are clear at every scale.

Level	What It Abstracts	Example
Architectural	Internal components of a subsystem; the system is viewed as a set of interacting services or layers	A three-tier architecture hides database details from the presentation layer
Design (Class)	Internal implementation of a class; the class is viewed through its public interface	A List interface hides whether the implementation is an array or linked list
Method	The step-by-step algorithm; the method is viewed by its signature and contract	sort(list) hides whether it uses quicksort, mergesort, or timsort
Data	The physical storage format; data is viewed through logical operations	A Map abstracts over hash tables, tree maps, or database lookups

2.3 Abstract Classes

An abstract class is a class that cannot be instantiated directly. It serves as a partial blueprint for subclasses, defining common attributes and methods while leaving one or more methods abstract — that is, declared but not implemented. Subclasses must provide concrete implementations of all abstract methods before they can be instantiated.

Abstract classes are used when a group of related classes share common structure and behaviour, but certain behaviours differ from one subclass to another. For example, an abstract class Shape might define a concrete method getColour() (the same for all shapes) and an abstract method area() (computed differently for each shape). Circle, Rectangle, and Triangle would each extend Shape and provide their own implementation of area().

In Java, an abstract class is declared with the abstract keyword. In Python, the abc module provides the ABC base class and the @abstractmethod decorator. In C++, a class with at least one pure virtual function (= 0) is abstract.

Language	Syntax for Abstract Class	Syntax for Abstract Method
Java	public abstract class Shape { }	public abstract double area();
Python	from abc import ABC, abstractmethod class Shape(ABC):	@abstractmethod def area(self): pass
C++	class Shape { /* ... */ };	virtual double area() = 0;

2.4 Interfaces

An interface is a purely abstract contract that specifies a set of method signatures without providing any implementation. Any class that implements an interface must supply concrete bodies for all of its methods. Interfaces define what an object can do without prescribing how it does it, representing the highest degree of abstraction in OOP.

Interfaces are particularly powerful for decoupling components. When code is written against an interface rather than a concrete class, any implementation can be substituted without modifying the calling code. This is the foundation of the Dependency Inversion Principle: high-level modules should depend on abstractions, not on concrete implementations.

A class can implement multiple interfaces, even in languages like Java that do not permit multiple inheritance of classes. This makes interfaces the preferred mechanism for defining capabilities or behaviours that cut across class hierarchies (e.g., Comparable, Serializable, Iterable).

Key Concept — Abstract Class vs Interface

An abstract class can contain both concrete and abstract methods, plus attributes; it represents an incomplete type in a hierarchy. An interface contains only method signatures (in traditional definitions) and represents a contract or capability. A class extends one abstract class but may implement many interfaces. Use abstract classes for shared implementation; use interfaces for shared contracts.

Feature	Abstract Class	Interface
Can contain concrete methods?	Yes	Java 8+: default methods Traditional: No
Can contain attributes?	Yes (any visibility)	Only public static final constants
Inheritance model	Single inheritance only (one parent class)	Multiple implementation (many interfaces)
Constructor?	Yes	No
When to use	Related classes sharing implementation code	Unrelated classes sharing a common capability or contract
UML notation	Class name in italics or «abstract» stereotype	«interface» stereotype above the class name

Session 3: Pillar 2 — Encapsulation

3.1 What is Encapsulation?

Encapsulation is the mechanism of bundling data (attributes) and the methods that operate on that data into a single unit — the class — while restricting direct access to the internal state from outside the class. It is the practical realisation of the information-hiding principle: external code interacts with an object only through its public interface (methods), never by directly reading or writing its internal attributes.

Encapsulation serves two fundamental purposes. First, it protects data integrity: by channelling all access through methods, the class can enforce rules (invariants) on its data. For example, a withdraw() method can check that the withdrawal amount does not exceed the balance, whereas direct access to the balance attribute would allow any code to set it to an invalid value. Second, it reduces coupling: because external code depends only on the public interface, the internal implementation can change freely without breaking anything elsewhere.

Key Concept — Encapsulation

Encapsulation bundles data and methods into a class while hiding the internal state behind a public interface. This protects data integrity (invariants are enforced by methods) and reduces coupling (internal changes do not affect external code). The mantra: make attributes private, expose behaviour through public methods.

3.2 Access Modifiers

Access modifiers are language keywords that control the visibility of class members (attributes and methods). They are the enforcement mechanism for encapsulation: by declaring an attribute as private, the programmer ensures that only the class itself can access it. Different languages offer different sets of access modifiers, but the core principle is the same.

Modifier	Java Keyword	Python Convention	C++ Keyword	Scope of Access
Public	public	no prefix	public:	Accessible from any class
Private	private	__ prefix (name mangling)	private:	Accessible only within the declaring class
Protected	protected	_ prefix (by convention)	protected:	Accessible within the class and its subclasses
Package (Default)	(no keyword)	N/A	N/A	Accessible within the same package

	(Java only)

The standard practice in well-designed OOP is to declare all attributes as private (or protected, if subclasses need direct access) and to provide public methods for controlled access. This pattern is so fundamental that it has a name: the accessor–mutator pattern (getters and setters).

3.3 Getters and Setters

A getter (accessor) is a public method that returns the value of a private attribute. A setter (mutator) is a public method that modifies the value of a private attribute, typically with validation logic to enforce invariants. Together, they provide controlled access to the internal state of an object.

For example, a BankAccount class might expose getBalance() (a getter) and setBalance(double newBalance) (a setter that rejects negative values). However, it is important not to create getters and setters mechanically for every attribute. If an attribute should never be modified externally, it should have no setter. If an attribute is purely internal, it should have neither getter nor setter. Thoughtful use of getters and setters is a hallmark of good encapsulation; mindless generation of both for every field undermines it.

Key Concept — Meaningful Encapsulation

Encapsulation is not merely making fields private and adding getters/setters. True encapsulation means exposing only the behaviour that external code genuinely needs. Ask: 'Does the outside world need to know this? Does it need to change this?' If not, provide neither getter nor setter.

3.4 Immutability

An immutable object is one whose state cannot be modified after construction. All attributes are set in the constructor and never changed thereafter. Immutability is the strongest form of encapsulation: if the state cannot change, there is no risk of external code corrupting it, no need for defensive copies, and no concurrency issues.

Java’s String class is the canonical example of an immutable type. Once a String is created, its content cannot be altered; methods like toUpperCase() return a new String rather than modifying the original. To create an immutable class, declare all attributes as private and final, provide no setters, and ensure that any mutable objects held as attributes are defensively copied.

Mutable Object	Immutable Object
State can change after creation	State is fixed at creation
Requires careful synchronisation in concurrent programs	Inherently thread-safe
Defensive copies may be needed when sharing	Can be freely shared without copying
Encapsulation enforced by access	Encapsulation guaranteed by

modifiers and validation	the absence of mutation

3.5 Encapsulation and Coupling

Coupling measures the degree to which one module depends on the internal details of another. Tight coupling means that changes in one module frequently require changes in another; loose coupling means modules can change independently. Encapsulation is the primary mechanism for achieving loose coupling in OOP.

When a class exposes only a stable public interface, other classes depend on that interface — not on the implementation behind it. The implementation can be refactored, optimised, or entirely rewritten without affecting any external code, provided the interface contract remains the same. This is why encapsulation is not merely a 'nice-to-have' convention but a structural property that makes large-scale software development feasible.

Session 4: Pillar 3 — Inheritance

4.1 What is Inheritance?

Inheritance is the mechanism by which one class (the subclass, child, or derived class) acquires the attributes and methods of another class (the superclass, parent, or base class). It establishes a hierarchical 'is-a' relationship: a subclass is a specialised kind of its superclass. The subclass inherits all non-private members of the superclass and may add new attributes and methods or override inherited methods to provide specialised behaviour.

Inheritance promotes code reuse by eliminating duplication: common attributes and methods are defined once in the superclass and automatically available in all subclasses. It also establishes a type hierarchy that enables polymorphism (Session 5), which is one of the most powerful features of OOP.

Key Concept — Inheritance

Inheritance allows a subclass to inherit attributes and methods from a superclass. It models the 'is-a' relationship: a Circle is a Shape, a SavingsAccount is a BankAccount. The subclass can extend (add new features) and override (replace behaviour) what it inherits.

4.2 Generalisation and Specialisation

Inheritance can be viewed from two complementary perspectives. Generalisation is the process of extracting common features from two or more classes into a shared superclass. Specialisation is the process of creating a new subclass that adds features specific to a particular kind of entity.

For example, consider three classes: Dog, Cat, and Fish. By observing that all three have attributes such as name and age and methods such as eat() and sleep(), a designer might generalise these into a superclass Animal. Conversely, when a designer realises that some animals can swim, they might specialise Animal into AquaticAnimal with additional attributes like waterType and methods like swim(). Both directions — bottom-up generalisation and top-down specialisation — are common in practice.

Direction	Process	Example
Generalisation (bottom-up)	Extract common features from existing classes into a new superclass	Dog, Cat, Fish → extract common features → Animal
Specialisation (top-down)	Create a new subclass that adds features specific to a subtype	Animal → specialise → AquaticAnimal (adds waterType, swim())

4.3 Method Overriding

Method overriding occurs when a subclass provides its own implementation of a method that is already defined in its superclass. The overriding method must have the same name, parameter list, and (in most languages) return type as the method it replaces. At runtime, the version of the method that executes is determined by the actual type of the object, not the declared type of the reference.

For example, a Shape superclass might define a method draw() with a default implementation. The Circle subclass overrides draw() to render a circle, and the Rectangle subclass overrides it to render a rectangle. If a variable is declared as Shape but holds a Circle object, calling draw() will invoke Circle's version. This behaviour is the basis of runtime polymorphism.

In Java, it is good practice to annotate overriding methods with @Override, which causes a compile-time error if the method does not actually override a superclass method. This prevents subtle bugs caused by accidental misspelling or incorrect parameter types.

Key Concept — Overriding vs Overloading

Overriding: a subclass replaces an inherited method with a new implementation (same name, same parameters, different class). Enables runtime polymorphism.
Overloading: multiple methods in the same class share a name but differ in parameter types or count. Resolved at compile time. These are distinct concepts that students frequently confuse.

4.4 The super Keyword

The super keyword (or its equivalent: base in C#, super() in Python) provides access to the superclass from within a subclass. It is used in two main contexts: calling the superclass constructor to initialise inherited attributes, and calling a superclass method that has been overridden when the subclass wants to extend (rather than completely replace) the inherited behaviour.

For example, if Animal has a constructor Animal(String name), the Dog subclass might call super(name) in its own constructor to initialise the inherited name attribute before setting Dog-specific attributes like breed. Similarly, if Dog overrides eat() but wants to include the general Animal eating behaviour, it can call super.eat() within its own eat() method and then add dog-specific logic.

4.5 Types of Inheritance

Type	Description	Support
Single	A class inherits from exactly one superclass	All OO languages
Multilevel	A chain of inheritance: A → B → C	All OO languages
Hierarchical	Multiple subclasses inherit from the	All OO languages

	same superclass	
Multiple	A class inherits from two or more superclasses simultaneously	C++: Yes Java/C#: No (use interfaces) Python: Yes (MRO)

Multiple inheritance — where a class has two or more direct superclasses — can cause the 'diamond problem': if both superclasses inherit from a common ancestor and override the same method, which version does the subclass inherit? C++ resolves this with virtual inheritance; Python uses a Method Resolution Order (MRO) algorithm; Java avoids the problem entirely by restricting classes to single inheritance, using interfaces as the mechanism for acquiring multiple contracts.

4.6 Composition Over Inheritance

A well-known principle in software design states: 'favour composition over inheritance'. Inheritance creates a tight coupling between superclass and subclass — changes to the superclass can break subclasses, and the subclass is permanently bound to the superclass hierarchy. Composition, on the other hand, achieves reuse by having a class contain instances of other classes as attributes, delegating behaviour to them.

For example, rather than creating a FlyingCar class that inherits from both Car and Aircraft (which creates a diamond problem and a fragile hierarchy), a better design might give Car an attribute of type FlyingCapability, which provides the flying behaviour through delegation. The Car 'has' a FlyingCapability rather than 'being' an Aircraft.

Key Concept — When to Inherit, When to Compose

Use inheritance when the 'is-a' relationship genuinely holds and the subclass truly is a specialised version of the superclass (e.g., Circle is a Shape). Use composition when the relationship is 'has-a' or 'uses-a' (e.g., Car has an Engine). When in doubt, prefer composition: it is more flexible and less fragile than inheritance.

Criterion	Inheritance	Composition
Relationship	'Is-a' (Circle is a Shape)	'Has-a' (Car has an Engine)
Coupling	Tight: subclass depends on superclass internals	Loose: the composing class depends only on the interface of the contained object
Flexibility	Fixed at compile time; cannot change parent at runtime	Can swap implementations at runtime via dependency injection
Reuse scope	Inherits all public/protected members (may inherit too much)	Delegates only the specific behaviour needed
UML notation	Solid line with hollow triangle (△)	Solid line with filled diamond (◆) for composition; hollow diamond (◇) for aggregation

Session 5: Pillar 4 — Polymorphism

5.1 What is Polymorphism?

Polymorphism, from the Greek for 'many forms', is the ability of objects of different classes to respond to the same method call in class-specific ways. It allows a single variable, parameter, or collection to hold objects of different types and to invoke the correct behaviour for each type automatically at runtime. Polymorphism is what transforms a rigid class hierarchy into a flexible, extensible system.

Consider a list of Shape references: [circle, rectangle, triangle]. Calling area() on each element invokes Circle.area(), Rectangle.area(), or Triangle.area() as appropriate, without the calling code needing to know or care about the specific type. This is the essence of polymorphism: one interface, many implementations.

Key Concept — Polymorphism

Polymorphism enables objects of different classes to be treated through a common interface. The correct method implementation is selected at runtime based on the actual type of the object (dynamic dispatch). This allows code to be written against abstractions rather than concrete types, dramatically improving flexibility and extensibility.

5.2 Types of Polymorphism

Type	Also Known As	Mechanism	Resolved At
Subtype polymorphism	Runtime polymorphism, inclusion polymorphism	Method overriding: a subclass provides its own implementation of an inherited method	Runtime (dynamic dispatch)
Ad-hoc polymorphism	Compile-time polymorphism	Method overloading: multiple methods with the same name but different parameter types	Compile time (static dispatch)
Parametric polymorphism	Generics / Templates	A class or method operates on a type parameter (e.g., List<T>, ArrayList<String>)	Compile time (type erasure in Java)

Of these three types, subtype polymorphism is the most important for OOP and the one most closely associated with the concept of polymorphism in general. It is the form that depends on inheritance

and method overriding, and it is the form that UML class diagrams and sequence diagrams are designed to represent.

5.3 Subtype Polymorphism in Detail

Subtype polymorphism works through two mechanisms: inheritance (or interface implementation) establishes the type relationship, and dynamic dispatch selects the correct method implementation at runtime. The process is as follows:

- A superclass or interface defines a method signature (e.g., Shape declares abstract double area()).
- Each subclass provides its own implementation of that method (e.g., Circle implements area() as πr^2; Rectangle implements it as width × height).
- Client code declares a variable of the superclass/interface type and assigns an object of any subclass to it (e.g., Shape s = new Circle(5)).
- When the method is called (s.area()), the Java Virtual Machine (or equivalent runtime) inspects the actual type of the object and invokes the correct implementation (Circle.area()).

This mechanism means that client code never needs to use instanceof checks or conditional logic to determine the type of an object. Adding a new subclass (e.g., Pentagon) requires no changes to existing client code — it simply creates a new class that implements the expected method. This property is known as the Open/Closed Principle: the system is open for extension (new subclasses) but closed for modification (existing code is unchanged).

5.4 Method Overloading (Ad-Hoc Polymorphism)

Method overloading allows a class to have multiple methods with the same name but different parameter lists (different number of parameters, different types, or both). The compiler determines which version to call based on the arguments provided at the call site. Unlike overriding, overloading is resolved at compile time and does not involve inheritance.

For example, a Calculator class might define: add(int a, int b), add(double a, double b), and add(int a, int b, int c). The compiler selects the appropriate version based on the argument types. Overloading improves readability by allowing conceptually similar operations to share a name, but it should be used judiciously — overloading methods that differ in subtle ways can cause confusion.

Feature	Method Overriding	Method Overloading
Relationship	Between superclass and subclass	Within the same class (or inherited methods)
Signature	Same name, same parameters	Same name, different parameters
Return type	Must be the same (or covariant)	May differ
Resolution	Runtime (dynamic dispatch)	Compile time (static dispatch)
Purpose	Specialise inherited behaviour	Provide multiple ways to invoke a conceptually similar operation

Annotation (Java)	@Override	None required

5.5 Generics (Parametric Polymorphism)

Generics allow classes, interfaces, and methods to operate on type parameters rather than concrete types. A generic class like List<T> can hold elements of any type: List<String>, List<Integer>, List<Shape>. The type is specified when the generic is used, and the compiler ensures type safety at compile time.

Without generics, a List that accepts Object elements would require unsafe casting when retrieving elements, with the risk of ClassCastException at runtime. Generics eliminate this risk by making the type explicit. They are essential for writing reusable, type-safe collection classes, utility methods, and frameworks.

In Java, generics use angle-bracket syntax: class Box<T> { private T content; }. In C++, the equivalent is templates: template<typename T> class Box { T content; }. Python, being dynamically typed, achieves similar flexibility without explicit generics, though type hints (e.g., list[str]) are increasingly used for clarity.

5.6 The Power of Polymorphism: Design Implications

Polymorphism is not merely a language feature; it is a design philosophy. It encourages developers to programme to interfaces (or abstract superclasses) rather than concrete implementations. This single practice has far-reaching consequences:

- Extensibility — New behaviours can be added by creating new classes that implement existing interfaces, without modifying any existing code (Open/Closed Principle).
- Testability — Dependencies can be replaced with mock or stub implementations during testing, because the code depends on interfaces, not concrete classes.
- Flexibility — Implementations can be swapped at runtime (e.g., through dependency injection or the Strategy pattern) without recompiling or restructuring.
- Reduced conditional logic — Polymorphism replaces chains of if–else or switch statements that check object types. Instead of asking 'what type are you?', the code simply calls a method and trusts the object to do the right thing.

Key Concept — Programme to Interfaces

The principle 'programme to an interface, not an implementation' is the cornerstone of flexible OOP design. Declare variables, parameters, and return types using the most abstract type that captures the required behaviour (e.g., List rather than ArrayList). This maximises the benefit of polymorphism.

Session 6: OOP Relationships and Design Principles

6.1 Relationships Between Classes

In any non-trivial system, classes do not exist in isolation. They are connected by relationships that define how objects interact, share data, and delegate responsibility. Understanding these relationships is essential for producing designs that are coherent, maintainable, and aligned with the UML model. The principal class relationships in OOP are: association, aggregation, composition, inheritance, realisation, and dependency.

Relationship	Nature	Strength	Example
Dependency	One class transiently uses another (e.g., as a method parameter)	Weakest	Printer uses Document (only during print())
Association	A structural link: one class holds a reference to another	Moderate	Teacher is associated with Student (teaches)
Aggregation	'Has-a' (shared): the whole contains parts that can exist independently	Moderate–Strong	Department has Employees (employees survive department)
Composition	'Owns-a' (exclusive): the whole owns parts whose lifecycle is tied to it	Strong	House owns Rooms (rooms cease to exist without the house)
Inheritance	'Is-a': subclass extends superclass	Strongest (structural)	Circle extends Shape
Realisation	'Implements': class fulfils an interface contract	Strong (contractual)	ArrayList implements List

6.2 Association in Code

An association between two classes is implemented by one or both classes holding a reference (attribute) to the other. The multiplicity of the association determines whether the attribute is a single reference or a collection.

UML Multiplicity	Java Implementation	Example
1 (exactly one)	private OtherClass field;	private Customer customer;

		// each Order has exactly one Customer
0..1 (optional)	private OtherClass field; // may be null	private Passport passport; // a Person may or may not have one
* or 0..* (many)	private List<OtherClass> field;	private List<Order> orders; // a Customer has zero or more Orders
1..* (one or more)	private List<OtherClass> field; // enforced by validation	private List<OrderLine> lines; // an Order must have at least one

6.3 Aggregation vs Composition in Code

Both aggregation and composition are implemented as an object holding a reference to another, but they differ in lifecycle management. In aggregation, the contained object is typically received via a constructor or setter and may be shared with other objects. In composition, the containing object creates the part in its own constructor and is responsible for its destruction.

Aspect	Aggregation (Shared)	Composition (Exclusive)
Lifecycle	Part exists independently; passed in from outside	Part is created by and dies with the whole
Ownership	Shared: the part may be referenced by multiple wholes	Exclusive: the part belongs to exactly one whole
Java pattern	Constructor receives the part as a parameter: this.employee = employee;	Constructor creates the part internally: this.room = new Room();
Destruction	Deleting the whole does NOT delete the parts	Deleting the whole automatically deletes all its parts
UML symbol	Hollow diamond (◇)	Filled diamond (◆)

6.4 SOLID Principles

The SOLID principles are five design guidelines that, when applied together, produce object-oriented systems that are easy to maintain, extend, and test. They were articulated by Robert C. Martin and have become a cornerstone of professional software development.

Letter	Principle	Summary	Consequence of Violation
S	Single Responsibility Principle (SRP)	A class should have only one reason to change	Classes become bloated and fragile; a change for one

			concern breaks another
O	Open/Closed Principle (OCP)	Classes should be open for extension but closed for modification	Adding new behaviour requires modifying and retesting existing code
L	Liskov Substitution Principle (LSP)	Objects of a superclass should be replaceable with objects of a subclass without breaking the program	Subclasses violate the contract of the superclass, causing unexpected behaviour
I	Interface Segregation Principle (ISP)	Clients should not be forced to depend on methods they do not use	Fat interfaces force classes to implement irrelevant methods, increasing coupling
D	Dependency Inversion Principle (DIP)	High-level modules should depend on abstractions, not on concrete implementations	High-level logic is tightly coupled to low-level details, making changes costly

Key Concept — SOLID Principles

SOLID is an acronym for five design principles: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. Together, they guide developers toward code that is modular, extensible, and robust. Violations of SOLID principles are the root cause of most 'code smells' in object-oriented systems.

6.5 Cohesion and Coupling

Two complementary metrics measure the quality of an object-oriented design: cohesion and coupling. Well-designed systems exhibit high cohesion and low coupling.

Metric	Definition	Goal	How to Achieve
Cohesion	The degree to which the members of a single class are related and focused on a single purpose	HIGH cohesion: each class does one thing well	Apply the Single Responsibility Principle; if a class has unrelated methods, split it
Coupling	The degree to which one class	LOW coupling: classes interact only	Apply encapsulation; programme to

	depends on the internal details of another class	through narrow interfaces	interfaces; use dependency injection

High cohesion means that every attribute and method in a class is closely related to the class's single responsibility. Low coupling means that classes communicate through minimal, stable interfaces. Together, these properties make a system easier to understand (each class is focused), easier to change (modifications are localised), and easier to test (classes can be tested in isolation).

Session 7: The Bridge from UML to Object-Oriented Code

7.1 Why a Bridge Matters

UML and OOP are two sides of the same coin. UML provides the visual modelling language for designing systems; OOP provides the programming paradigm for implementing them. The 'bridge' between UML and OOP is the systematic process of translating UML diagram elements into working code. When this bridge is well understood, UML diagrams become directly implementable blueprints rather than abstract illustrations; conversely, existing code can be reverse-engineered into UML for analysis and documentation.

This session provides a comprehensive, element-by-element mapping from UML constructs to their OOP (Java) equivalents, covering classes, attributes, operations, relationships, and behavioural elements.

Key Concept — The UML–OOP Bridge

UML models and OOP code represent the same system at different levels. UML is the design-time representation; code is the runtime representation. The bridge between them is a systematic, almost mechanical translation: every UML element has a direct code counterpart, and vice versa.

7.2 Translating a UML Class to Code

The most fundamental translation is from a UML class box to a class declaration in code. Each compartment of the class box maps to a specific code construct.

UML Element	Class Box Location	Java Code Equivalent	Example
Class name	Top compartment	class ClassName { }	class BankAccount { }
Abstract class (italicised name)	Top compartment with «abstract»	abstract class Name { }	abstract class Shape { }
Interface («interface»)	Top compartment with stereotype	interface Name { }	interface Comparable { }
Attribute (− name : String)	Middle compartment	private String name;	private String name;
Attribute with default (− balance : double = 0.0)	Middle compartment	private double balance = 0.0;	private double balance = 0.0;
Static attribute (underlined)	Middle compartment	private static int count;	private static int count;
Operation (+ deposit(amt : double)	Bottom compartment	public void deposit(double amt)	public void deposit(double amt)

: void)			{ ... }
Abstract operation (italicised)	Bottom compartment	public abstract double area();	public abstract double area();
Static operation (underlined)	Bottom compartment	public static int getCount()	public static int getCount() { ... }

7.3 Translating Visibility

UML visibility markers map directly to Java access modifiers. This translation is straightforward but critical for maintaining the encapsulation designed into the UML model.

UML Symbol	UML Name	Java Keyword	Python Convention
+	Public	public	No prefix (e.g., name)
−	Private	private	Double underscore (e.g., __name)
#	Protected	protected	Single underscore (e.g., _name)
~	Package	(default / no keyword)	N/A

7.4 Translating Relationships to Code

UML relationships translate into specific code patterns. This section provides a systematic mapping from each UML relationship type to its Java implementation, with fully worked examples.

7.4.1 Association

A UML association between two classes translates to one or both classes holding a reference (field) to the other. The multiplicity determines the field type: a multiplicity of '1' maps to a single reference; '*' maps to a collection.

UML Association	Java Code

Customer 1 — * Order
(a Customer has many Orders)
	// In Customer class:
	private List<Order> orders;
	
	// In Order class:
	private Customer customer;

Student * — * Module
(many-to-many)
	// In Student class:
	private List<Module> modules;
	
	// In Module class:
	private List<Student> students;

Person 1 — 0..1 Passport (optional association)	// In Person class: private Passport passport; // may be null

7.4.2 Aggregation

Aggregation translates to a field holding a reference to the part, but the part is created externally and passed in. The containing class does not create or destroy the part.

UML Aggregation	Java Code

Department ◇— * Employee
(Department aggregates Employees)
	public class Department {
	  private List<Employee> employees;
	
	  public void addEmployee(Employee e) {
	  employees.add(e);
	  }
	  // Employee is NOT created here;
	  // it exists independently
	}

7.4.3 Composition

Composition translates to the whole creating and owning its parts. The parts are typically instantiated within the constructor and have no independent existence outside the whole.

UML Composition	Java Code

Order ◆— 1..* OrderLine
(Order is composed of OrderLines)
	public class Order {
	  private List<OrderLine> lines;
	
	  public Order() {
	  this.lines = new ArrayList<>();
	  }
	
	  public void addLine(String product, int qty) {
	  lines.add(new OrderLine(product, qty));
	  // OrderLine is CREATED by Order
	  }
	  // When Order is deleted, its
	  // OrderLines are also deleted
	}

7.4.4 Inheritance (Generalisation)

The UML generalisation arrow (solid line with hollow triangle) maps directly to the extends keyword in Java (or the : syntax in C++, or parenthetical syntax in Python).

UML Generalisation	Java Code

Circle –▷ Shape
(Circle inherits from Shape)
	public class Shape {
	  protected String colour;
	  public double area() { return 0; }
	}
	
	public class Circle extends Shape {
	  private double radius;
	
	  @Override
	  public double area() {
	  return Math.PI * radius * radius;
	  }
	}

7.4.5 Realisation (Interface Implementation)

The UML realisation arrow (dashed line with hollow triangle) maps to the implements keyword in Java.

UML Realisation	Java Code

Student - -▷ «interface» Comparable
(Student implements Comparable)
	public interface Comparable<T> {
	  int compareTo(T other);
	}
	
	public class Student implements
	Comparable<Student> {
	  private String name;
	  private double gpa;
	
	  @Override
	  public int compareTo(Student other) {
	  return Double.compare(this.gpa, other.gpa);
	  }
	}

7.4.6 Dependency

A dependency is the weakest relationship: one class uses another transiently (e.g., as a method parameter, local variable, or return type) without storing a permanent reference.

UML Dependency	Java Code

Printer - - -> Document
(Printer uses Document)
	public class Printer {
	  public void print(Document doc) {
	  // uses doc only within this method
	  System.out.println(doc.getContent());

```
    }
    // no Document field stored
}
```

7.5 Complete UML-to-Code Reference

The following table provides a consolidated, at-a-glance reference for translating every major UML class-diagram element into Java code.

UML Construct	Java Equivalent	Key Notes
Class	class Name { }	One file per public class
Abstract class	abstract class Name { }	Cannot be instantiated
Interface	interface Name { }	All methods implicitly public abstract
+ (public)	public	Accessible everywhere
− (private)	private	Accessible only in this class
# (protected)	protected	Accessible in this class + subclasses
~ (package)	(default)	Accessible within the same package
Attribute	private Type name;	Prefer private with getters/setters
Operation	public ReturnType name(params)	Method signature maps directly
Association (1)	private OtherClass field;	Single reference
Association (*)	private List<Other> field;	Collection for * multiplicity
Aggregation ◇	field = received param	Part passed in; shared ownership
Composition ◆	field = new Part()	Part created internally; exclusive ownership
Generalisation △	class Child extends Parent	Single inheritance in Java
Realisation △ (dashed)	class C implements I	Multiple interfaces allowed
Dependency -->	method parameter or local var	No stored reference
Static (underlined)	static keyword	Belongs to class, not instance
Abstract (italic)	abstract keyword	No body; subclass must implement

Session 8: Translating Behavioural UML to Code

8.1 From Sequence Diagrams to Method Calls

Sequence diagrams model the runtime interactions between objects. Each message arrow in a sequence diagram corresponds to a method call in code. The translation is direct: the sending object calls a method on the receiving object, passing any required parameters, and (optionally) receiving a return value.

Sequence Diagram Element	Code Equivalent	Example
Synchronous message A → B: calculate(x)	Object A calls B.calculate(x) and waits for it to return	double result = b.calculate(x);
Return message B ---> A: result	The return value from B.calculate(x)	return result; // (in B's calculate method)
Self-message A → A: validate()	Object A calls its own method	this.validate();
Create message A → «create» B	Object A instantiates B	B b = new B();
Conditional (alt fragment)	if–else block	if (condition) { ... } else { ... }

Loop fragment	while or for loop	while (hasNext) {
		process(item);
		}

When translating a sequence diagram, work through each message from top to bottom. For each message, identify the sender, receiver, method name, and parameters. Write the corresponding method call in the sender's code, and ensure the receiver's class has the method defined. Combined fragments (alt, loop, opt) translate to standard control-flow structures (if–else, while, if).

8.2 From State Machine Diagrams to Code

State machine diagrams model how an individual object transitions between states in response to events. Translating a state machine to code involves representing the current state, handling events that trigger transitions, and executing entry/exit/transition actions.

State Machine Element	Code Pattern	Example

State	An enum value or constant representing the current state	enum OrderState {
		PENDING, CONFIRMED,
		SHIPPED, DELIVERED
		}

Transition
event [guard] / action
	Method that checks the current state, evaluates the guard, performs	if (state == PENDING
		&& paymentValid) {
		issueConfirmation();

	the action, and updates the state	state = CONFIRMED; }
Entry action entry / startTimer()	Code executed immediately after setting the new state	state = ACTIVE; startTimer(); // entry action

Exit action
exit / stopTimer()
	Code executed immediately
	before leaving the current state
		stopTimer(); // exit action
		state = NEXT_STATE;

There are two common code patterns for implementing state machines. The simple approach uses an enum for states and switch/if-else statements for event handling. The more sophisticated State Pattern (a Gang of Four design pattern) uses polymorphism: each state is a separate class implementing a common State interface, and state-specific behaviour is handled by method overriding. The State Pattern is preferable when the state machine is complex or likely to evolve.

Key Concept — State Pattern

The State Pattern encapsulates state-specific behaviour in separate classes that implement a common interface. The context object delegates behaviour to its current state object. To transition, the context switches its state reference to a different state object. This avoids sprawling switch statements and makes adding new states trivial.

8.3 From Activity Diagrams to Code

Activity diagrams model workflows and processes. Their translation to code is typically the most flexible, as the same activity diagram can be implemented using different code structures depending on the context.

Activity Diagram Element	Code Pattern
Action node	A method call or a block of statements
Decision node [guard conditions]	if–else if–else chain or switch statement
Merge node	The point where if/else branches reconverge (no explicit code; it's the end of the if block)
Fork bar (concurrent flows)	Thread creation, ExecutorService, or CompletableFuture for parallel execution
Join bar (synchronise flows)	Thread.join(), Future.get(), or CountDownLatch.await()
Loop (via decision + merge)	while loop or for loop
Swim lanes	Different classes or services; each lane maps to a class whose methods implement the actions in that lane

Swim lanes are particularly significant for the bridge to code: each lane typically corresponds to a class (or service) in the implementation. The actions within a lane become methods on that class, and flow edges crossing lane boundaries become method calls between classes — which are exactly the messages that appear in the corresponding sequence diagram.

Session 9: Worked Example — From UML to Complete Implementation

9.1 Problem Description

To bring together all the concepts in this resource, this session walks through a complete worked example: modelling and implementing a simple Library Management System. The system must support the following requirements: a library holds a collection of books; members can borrow and return books; each book tracks its availability status; and members have a borrowing limit.

9.2 Step 1: Identifying Classes (from UML Class Diagram)

From the problem description, the key nouns suggest the following classes: Library, Book, Member, and Loan (to record a borrowing transaction). Applying the four pillars:

- Abstraction — Each class captures only the essential features for this system. Book has title, author, ISBN, and availability — not publisher address or printing history.
- Encapsulation — All attributes are private; access is through public methods (e.g., getTitle(), setAvailable()).
- Inheritance — If the system distinguished between physical books and e-books, Book could be an abstract superclass with PhysicalBook and EBook subclasses.
- Polymorphism — A common interface Borrowable could be implemented by both Book and Journal, allowing the borrowing logic to work with any borrowable item.

9.3 Step 2: Defining Relationships

Relationship	UML	Code

Library ◆— * Book (composition)	Filled diamond at Library end; * at Book end	class Library {
		  private List<Book> books
		  = new ArrayList<>();
		}

Library ◇— * Member (aggregation)	Hollow diamond at Library end; * at Member end	class Library {
		  private List<Member>
		members;
		  void register(Member m) {
		  members.add(m);
		  }
		}

Member 1 — * Loan (association)	1 at Member end; * at Loan end	class Member {
		  private List<Loan> loans;
		}

Loan * — 1 Book (association)	* at Loan end; 1 at Book end	class Loan { private Book book;

```
}
```

Note the distinction: Library composes Books (if the library is dissolved, its books are removed from the system) but aggregates Members (members can exist outside the library context). Loan associates Member and Book: each loan references exactly one member and one book.

9.4 Step 3: Applying Visibility and Encapsulation

Class	Attributes (private)	Key Methods (public)
Book	− title : String − author : String − isbn : String − available : boolean	+ getTitle() : String + isAvailable() : boolean + setAvailable(status : boolean)
Member	− name : String − memberId : String − loans : List<Loan> − maxLoans : int = 5	+ borrow(book : Book) : Loan + returnBook(loan : Loan) + canBorrow() : boolean
Loan	− book : Book − borrowDate : LocalDate − returnDate : LocalDate	+ getBook() : Book + isOverdue() : boolean
Library	− books : List<Book> − members : List<Member>	+ addBook(b : Book) + register(m : Member) + findAvailable() : List<Book>

9.5 Step 4: State Machine for Book

The Book object has a simple two-state lifecycle that illustrates state machine concepts:

- States: Available, On Loan.
- Transition: Available → On Loan, triggered by event borrow [memberCanBorrow] / createLoan().
- Transition: On Loan → Available, triggered by event return / closeLoan().

In code, the available boolean attribute implements this state machine. The borrow() method checks the guard (isAvailable() && member.canBorrow()), performs the action (creates a Loan), and transitions the state (setAvailable(false)). The returnBook() method reverses the transition.

9.6 Step 5: Sequence Diagram to Code

A sequence diagram for the 'Borrow Book' use case would show the following message flow: Member sends borrow(book) to Library; Library sends isAvailable() to Book; if true, Library sends canBorrow() to Member; if true, Library creates a new Loan, sends setAvailable(false) to Book, and returns the Loan to Member.

Each message translates to a method call. The alt fragment (checking availability and borrowing limit) becomes an if statement. The create message becomes new Loan(member, book, LocalDate.now()). This direct, traceable mapping from diagram to code is the bridge from UML to OOP in action.

Key Concept — Traceability in Practice

In the worked example, every class, attribute, method, and relationship in the code traces directly back to a UML diagram element. Use case → activity diagram → class diagram → state machine → sequence diagram → code. This chain of traceability is the hallmark of disciplined, model-driven development.

9.7 Summary of the Four Pillars in the Example

Pillar	Where It Appears in the Example
Abstraction	Each class captures only the essential attributes and methods needed by the system. Book does not model physical dimensions, shelf location, or printing details — only what is relevant to borrowing.
Encapsulation	All attributes are private. The available state of a Book is changed only through controlled methods (borrow/return) that enforce business rules (e.g., cannot borrow an unavailable book).
Inheritance	If the system is extended to include EBook and PhysicalBook, they would inherit from Book. Common behaviour (getTitle(), isAvailable()) is defined once in the superclass.
Polymorphism	A Borrowable interface could allow both Book and Journal to be borrowed using the same borrow() logic. The Member class would call borrow(Borrowable item) without knowing the specific type.

References and Further Reading

- Bloch, J. (2018) Effective Java. 3rd edn. Boston: Addison-Wesley.
- Martin, R.C. (2009) Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River, NJ: Prentice Hall.
- Martin, R.C. (2018) Clean Architecture: A Craftsman's Guide to Software Structure and Design. Upper Saddle River, NJ: Prentice Hall.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.
- Fowler, M. (2004) UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd edn. Boston: Addison-Wesley.
- Booch, G., Rumbaugh, J. and Jacobson, I. (2005) The Unified Modeling Language User Guide. 2nd edn. Boston: Addison-Wesley.
- Larman, C. (2004) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd edn. Upper Saddle River, NJ: Prentice Hall.
- Sommerville, I. (2016) Software Engineering. 10th edn. Harlow: Pearson.
- Pressman, R.S. and Maxim, B.R. (2020) Software Engineering: A Practitioner's Approach. 9th edn. New York: McGraw-Hill.
- Object Management Group (2017) Unified Modeling Language Specification, Version 2.5.1. Available at: https://www.omg.org/spec/UML/ (Accessed: February 2026).
- Oracle (2024) The Java Tutorials: Object-Oriented Programming Concepts. Available at: https://docs.oracle.com/javase/tutorial/java/concepts/ (Accessed: February 2026).