A **package diagram** is a type of structural diagram in **UML (Unified Modeling Language)** used in **software design and architecture** to show how a system is organized into **packages**—logical groupings of related classes, components, or subsystems.

Package diagram, a kind of structural diagram, shows the arrangement and organization of model elements in middle to large scale project. Package diagram can show both structure and dependencies between sub-systems or modules, showing different views of a system, for example, as multi-layered (aka multi-tiered) application - multi-layered application model.

Purpose of Package Diagrams

Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.

- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- A package is a collection of logically related UML elements.
- Packages are depicted as file folders and can be used on any of the UML diagrams.

What Is a Package?

A **package** is a high-level container that groups related elements together. Think of it like a folder that organizes code components by functionality, domain, or layer.

Examples:

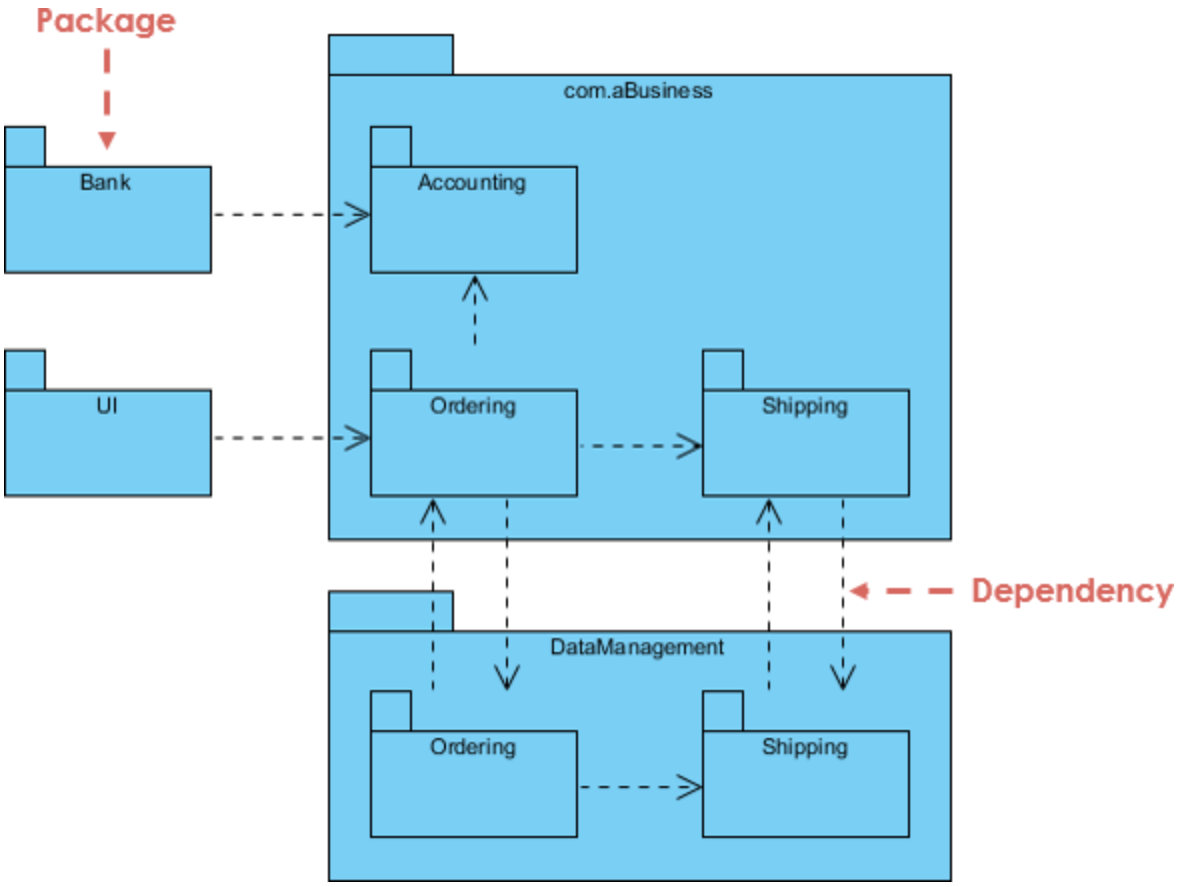
- ui, businessLogic, dataAccess
- authentication, payment, notifications

Package Diagram at a Glance

Package diagram is used to simplify complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements.

The diagram below is a business model in which the classes are grouped into packages:

- Packages appear as rectangles with small tabs at the top.
- The package name is on the tab or inside the rectangle.
- The dotted arrows are dependencies.
- One package depends on another if changes in the other could possibly force changes in the first.

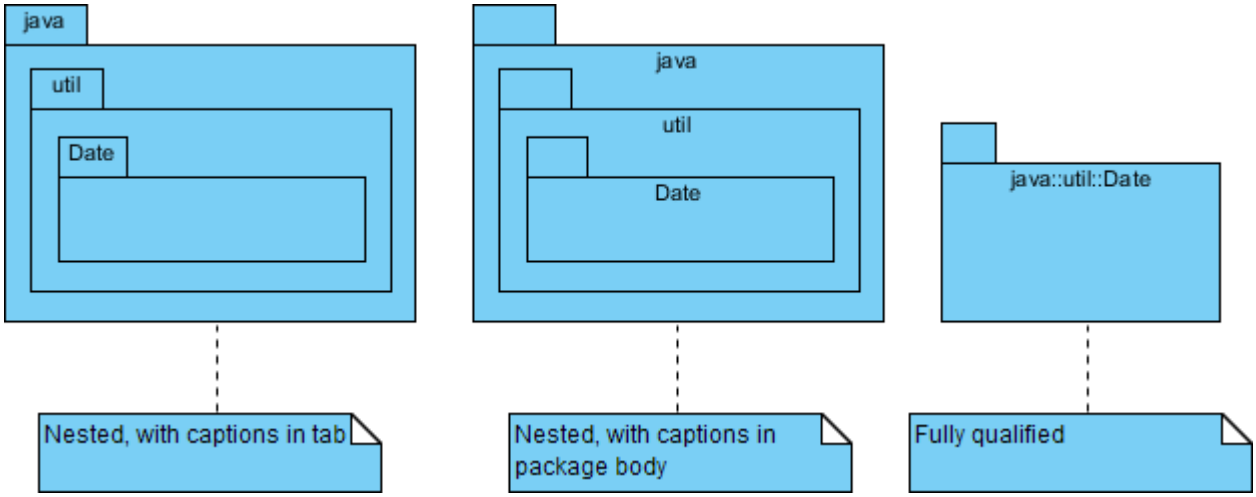


Basic Concepts of Package Diagram

Package diagram follows hierarchal structure of nested packages. Atomic module for nested package are usually class diagrams. There are few constraints while using package diagrams, they are as follows.

- Package name should not be the same for a system, however classes inside different packages could have the same name.
- Packages can include whole diagrams, name of components alone or no components at all.

Packages can be represented by the notations with some examples shown below:

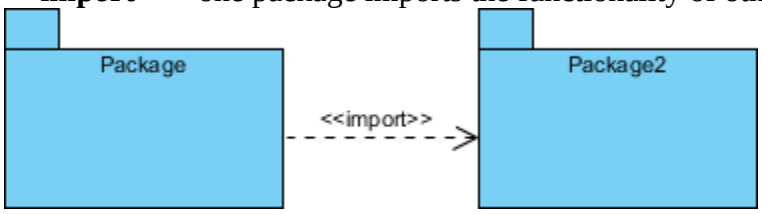


Package Diagram - Dependency Notation

There are two sub-types involved in dependency. They are <<import>> & <<access>>. Though there are two stereotypes users can use their own stereotype to represent the type of dependency between two packages.

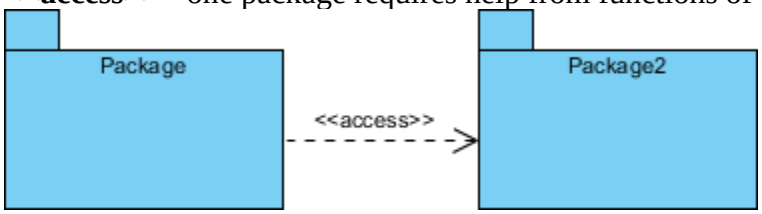
Package Diagram Example - Import

<<import>> - one package imports the functionality of other package



Package Diagram Example - Access

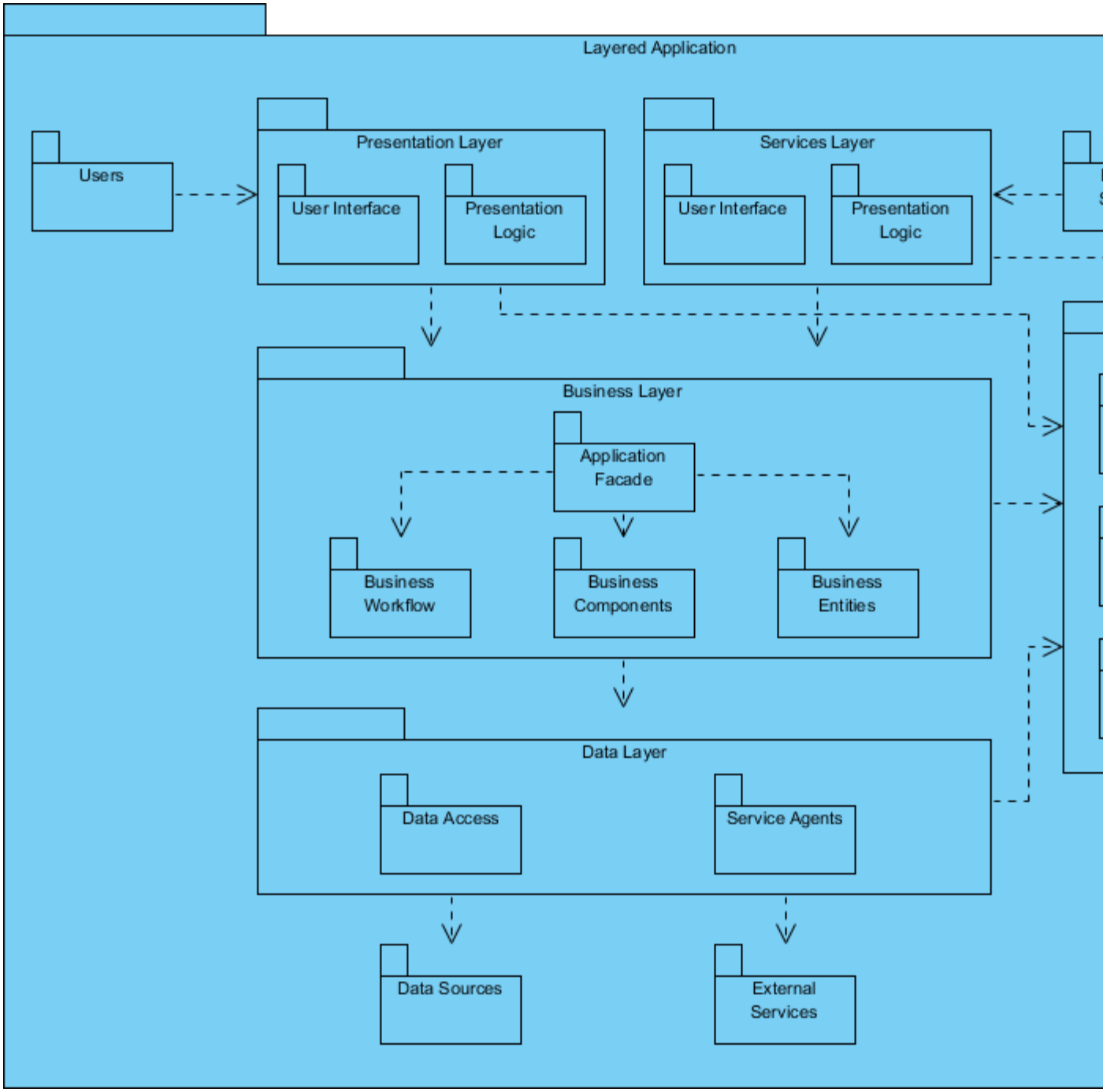
<<access>> - one package requires help from functions of other package.



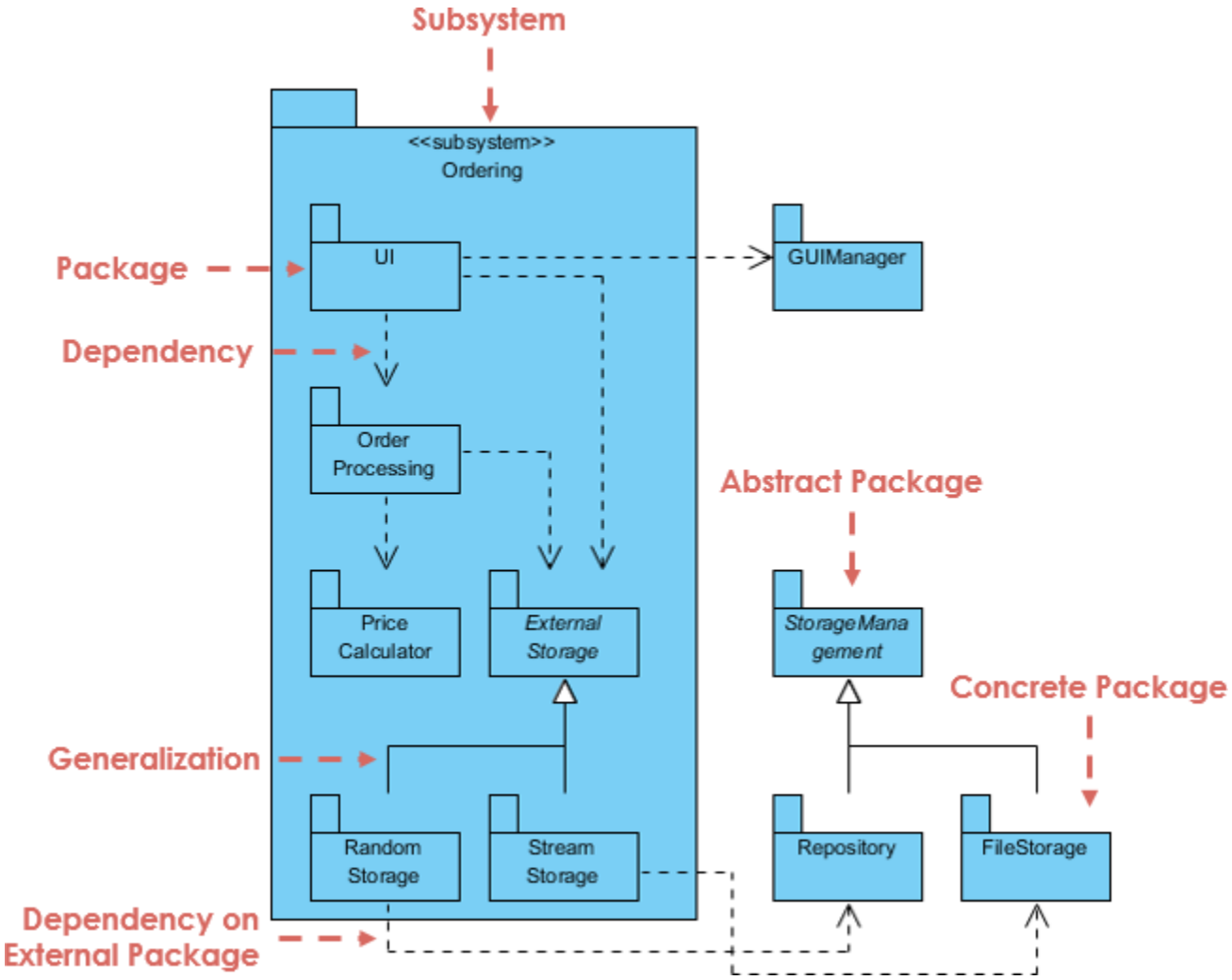
Modeling Complex Grouping

A package diagram is often used to describe the hierarchical relationships (groupings) between packages and other packages or objects. A package represents a namespace.

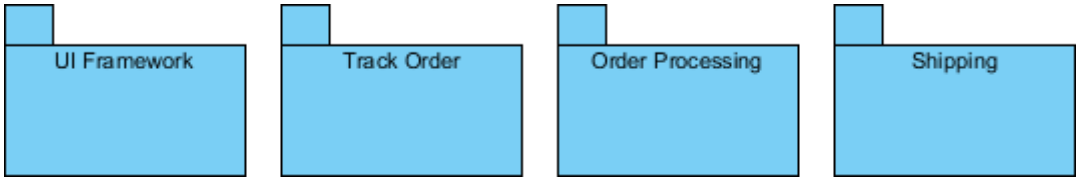
Package Diagram Example - Layering Structure


Layered Application
Users
Presentation Layer
User Interface
Presentation Logic
Services Layer
User Interface
Presentation Logic
Business Layer
Application Facade
Business Workflow
Business Components
Business Entities
Data Layer
Data Access
Service Agents
Data Sources
External Services

Package Diagram Example - Order Subsystem



Package Diagram Example - Order Processing System



Package Diagram Example - Order Processing System

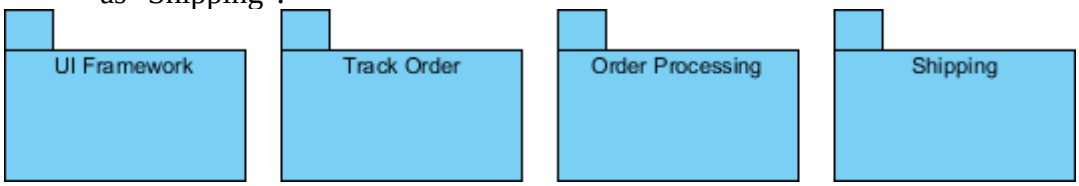
Order Processing System - The Problem Description

We are going to design package diagram for "Track Order" scenario for an online shopping store. Track Order module is responsible for providing tracking information for the products ordered by customers. Customer types in the tracking serial number, Track Order modules refers the system and updates the current shipping status to the customer.

Based on the project Description we should first identify the packages in the system and then related them together according to the relationship:

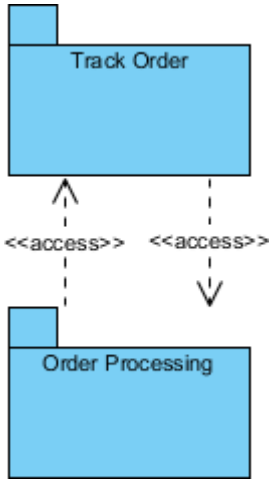
Identify the packages of the system

- There is a track order module, it has to talk with other module to know about the order details, let us call it "Order Details".
- Next after fetching Order Details it has to know about shipping details, let us call that as "Shipping".

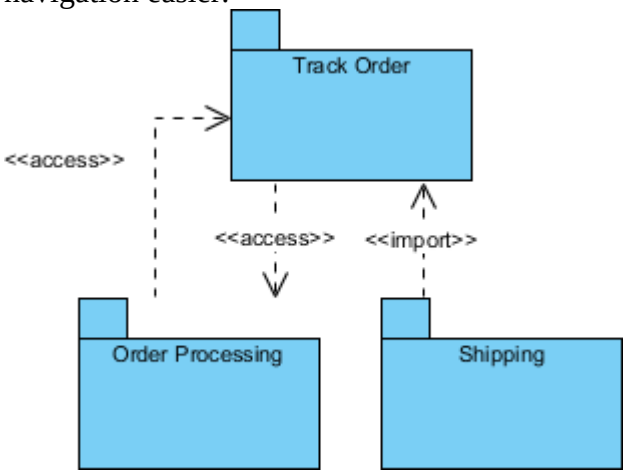


Identify the dependencies in the System

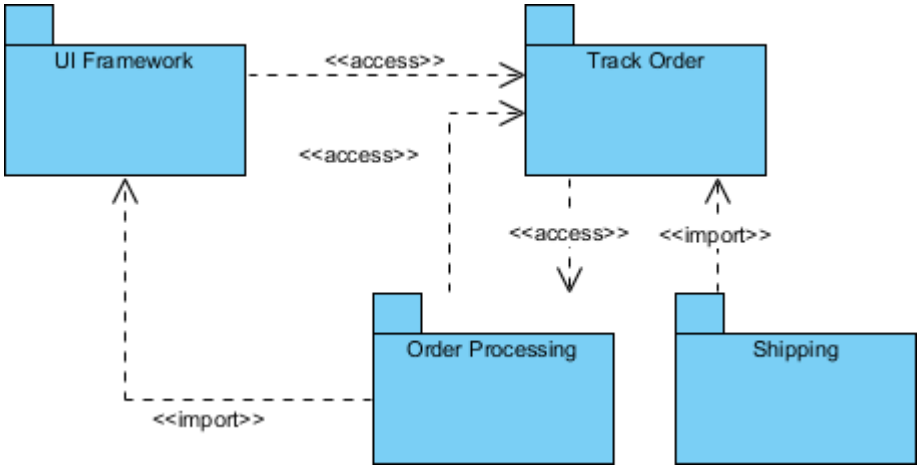
- Track order should get order details from "Order Details" and "Order Details" has to know the tracking info given by the customer. Two modules are accessing each other which suffices <<access>> dual dependency



- To know shipping information, "Shipping" can import "Track Order" to make the navigation easier.



- Finally, Track Order dependency to UI Framework is also mapped which completes our Package Diagram for Order Processing subsystem.



In **UML package diagrams**, an **abstract package** is a package that **cannot be instantiated directly** and is intended to be **inherited or extended by other packages**.
It works similarly to an **abstract class**, but at a higher (architectural) level.

✓ What Is an Abstract Package?

An **abstract package** is a package that represents a **generic, incomplete, or conceptual grouping** of functionality.
It defines **common structure, interfaces, or dependencies** that other (concrete) packages will specialize.

It is shown with:

- The package name **in italics**, or
- The stereotype **«abstract»**

Example notation:

«abstract»
PaymentCore

or

PaymentCore (in italics)

✓ Why Use an Abstract Package?

Abstract packages are used when you want to:

1. Define shared architectural rules

Example: A base "CoreServices" package may declare interfaces that all service modules must follow.

2. Enforce consistent structure across multiple packages

Like:

- AbstractPersistence → extended by SQLPersistence, NoSQLPersistence

3. Organize common behavior or dependencies

Abstract packages can define:

- Common interfaces
- Shared classes
- Shared dependency constraints

But they are **not complete** and cannot be used as a standalone module.

✓ Example Scenario

Suppose you design a payment system.

Abstract Package
«abstract»
PaymentMethod

This abstract package defines general interfaces like:

- processPayment()
- refund()
- validate()

Specialized Packages
CreditCardPayment → PaymentMethod (abstract)
PayPalPayment → PaymentMethod (abstract)
CryptoPayment → PaymentMethod (abstract)

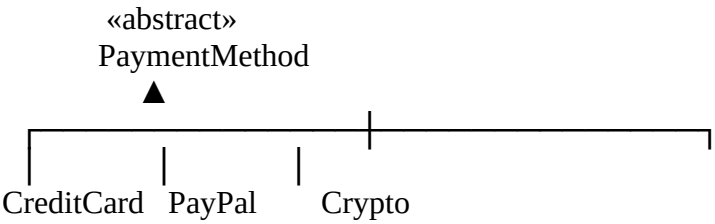
Each concrete package provides actual implementations.

✓ How Abstract Packages Appear in UML

Notation

- Package name in *italics*
- Or stereotype: «abstract»
- May include abstract classes or interfaces
- Usually connected via **generalization arrow** (solid line + hollow triangle)

Example:



✓ Key Characteristics of an Abstract Package

Feature	Description
Cannot be instantiated	Same as abstract classes
Provides template structure	Defines common interfaces or dependencies
Extended by concrete packages	Concrete packages inherit or implement its elements
Helps with architectural reuse	Encourages consistency across modules

---

✓ In Simple Terms

An **abstract package** is the architectural "blueprint" for a group of related packages. It defines **what should exist**, but not the fully working implementation.

If you want, I can also provide:
✓ a diagram example
✓ differences between abstract package vs abstract class
✓ when to use abstract packages in layered architecture

A **concrete package** in UML is a package that **can be used as-is**, meaning it contains **complete, fully defined elements** (such as classes, interfaces, components) and **can be instantiated or deployed** as part of a real system.

It is the opposite of an **abstract package**, which represents a conceptual or partially defined grouping.

✓ What Is a Concrete Package?

A **concrete package**:

- Represents a **fully implemented** module of the system

- Contains **non-abstract classes, interfaces, components, or sub-packages**
- Can be **compiled, instantiated, or executed**
- Implements or specializes any structure defined by an **abstract package**
- Plays a **real, operational role** in the architecture

✓ Notation

Concrete packages use **normal package notation**:

```
+------------------+
|   UserService    |
+------------------+
```

They **do NOT** use italics and **do NOT** have the stereotype «abstract».

✓ Example

Imagine an abstract package:

«abstract»
NotificationService

This abstract package might define interfaces like:

- send()
- schedule()
- validate()

Concrete packages that implement it

EmailNotification → implements NotificationService
SMSNotification → implements NotificationService
PushNotification → implements NotificationService

These **concrete packages** contain actual working classes such as:

- EmailSender
- SMSSender
- PushAPIAdapter

Each concrete package turns the abstract concepts into real, usable functionality.

✓ Characteristics of a Concrete Package

Feature	Description
Complete implementation	Contains usable classes/components
Instantiable	Can be executed or compiled
Implements an abstract package	Often refines templates or interfaces
Operational	Used directly in the system
Non-italic name	Appears normal in UML diagrams

✓ Concrete vs. Abstract Package (Quick Comparison)

Aspect	Abstract Package	Concrete Package
Completeness	Incomplete; conceptual	Fully implemented; usable
Instantiation	Cannot be instantiated	Can be instantiated and executed
Purpose	Define shared rules or structure	Provide real functionality
Notation	Italics or «abstract»	Normal package symbol

✓ Simple Real-World Analogy

- **Abstract package** = Blueprint for a type of building
- **Concrete package** = An actual constructed building that people can use

What a Package Diagram Shows

✓ 1. Dependencies between packages

It visualizes how packages rely on each other. Dependency is shown with a **dashed arrow** →.

Example:

UI → BusinessLogic → DataAccess

✓ 2. Package hierarchy

Packages can contain:

- Sub-packages
- Classes
- Components

This helps in understanding modular structure.

✓ 3. System layering

Common architecture layers shown in package diagrams:

- Presentation Layer
- Application/Domain Layer
- Data Layer

✓ 4. Encapsulation of code

Packages make it easier to understand what should be public or private.

Why Package Diagrams Matter

1. Manage complexity

Large systems become easier to understand when divided into logical sections.

2. Enforce modularity

Clear package boundaries help:

- isolate changes
- prevent tight coupling
- improve maintainability

3. Support layered architecture

Great for representing:

- MVC
- Clean Architecture
- Hexagonal Architecture

4. Aid communication

Helps developers and architects get a high-level overview before diving into class details.

Common Notations in a Package Diagram

Notation	Meaning
Folder icon	Represents a package
Dashed arrow	Dependency ("uses")
Solid line	Association
«import»	One package imports elements from another
«access»	One package accesses public members of another

Example of a Simple Package Diagram

Interpretation:

- UI depends on Business Logic
- Business Logic depends on Data Access
- UI should NOT directly access Data Access (enforced layering)

When To Use a Package Diagram

Use it when you need to:

- design system architecture at a high level
- break a system into modules or layers
- document package responsibilities
- manage dependencies across teams or components
