```
Package
=======
A package is used to encapsulate/group similar types of classes,
interfaces and sub-packages.

A Package is nothing but a physical folder structure where code is
structured according to usage.

It is a good practice to arrange code in packages and sub-packages

Need of Packages
----------------
Here are the key points on the importance of using packages:

1. Organizes Classes - Packages group related classes logically.
For ex:
- All entities can be put into ecom.app.entities package
- All database related files can be put into ecom.app.dbutils package

2. Avoids Naming Conflicts - Pacakges prevent naming collisions between
classes.
Ex:
- Suppose we wish to create one Utils.java for Orders as well as Sales
modules
- So we create one Utils.java each in ecom.app.order package, and in
ecom.app.sales packages.
- So the fully qualified names will be ecom.app.order.Utils and
ecom.app.sales.Utils

3. Access Control - Packages control class and data visibility with
access modifiers.

class OrderCheck {
      int orderId;
}

Now OrderCheck class will be visible only in its package as it is package
private

4. Reusability - Packages promote code reuse across multiple projects.

5. Easier Maintenance - Packages simplify code management and updates.

Ex: Make searching/locating and usage of classes, interfaces,
enumerations and annotations easier

Features
--------
- The package statement must be the 1st statement within a
class/interface.
- There can be only 1 package statement mentioned in any file.
- It is a good practice to avoid default package.

Naming
```

```
------
- The packages are usually named in all smaller case. For ex:
ecom.app.controllers
- However it depends on project's preferences

Import
======
- We need to import a class in our code before using it
- This is done with import keyword.
- We can import n number of classes, interfaces, enums and static
variables and methods
- If the classes are within the same package we don't have to import them
- The content of java.lang package are imported implicitly
- So, in order to use System, String, Integer, Exception we don't need to
import them.
- All other classes that are not in java.lang have to be improted

Types of import
---------------
These are the types of imports we can do.

 1. Single Type Import
- Syntax: `import packageName.ClassName;`
- Description: Imports a specific class, interface, or enum from a
package.
- Example: `import java.util.ArrayList;`

 2. Wildcard Import
- Syntax: `import packageName.*;`
- Description: Imports all classes, interfaces, and enums from a package.
- Use star only when we need 4 or more classes/interfaces from a package.
- Example: `import java.util.*;`

 3. Static Import
- Syntax: `import static packageName.ClassName.staticMember;`
- Description: Imports a specific static member (method or field) of a
class.
- Example: `import static java.lang.Math.PI;`

 4. Static Wildcard Import
- Syntax: `import static packageName.ClassName.*;`
- Description: Imports all static members of a class.
- Example: `import static java.lang.Math.*;`

 5. Importing Enums
- Syntax: `import packageName.EnumName;`
- Description: Enums can be imported just like any other class or
interface.
- Example: `import java.time.DayOfWeek;`

 6. Fully Qualified Class Name Usage
- Syntax: Use the full package path when referring to the class.
- Description: Directly uses the class with its full package name,
avoiding the need for an import.
```

```
- Example:
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date(System.currentTimeMillis());

 Summary
- Use single type import for specific types.
- Use wildcard import to import all types from a package.
- Use static import for specific static members.
- Use static wildcard import to import all static members from a class.
- Use fully qualified class name to avoid ambiguity between classes with
the same name from different packages.

java.lang
---------
- java.lang contains the core classes of the Java Library
- For example, the following classes are a part of java.lang package
      - All wrapper classes (Integer, Double etc)
      - String handling classes (String, StringBuilder etc)
      - System, Scanner classes etc
- java.lang package is implicitly imported by the Java Runtime System
- That is why we haven't imported them in any file.
-----X----X----X----X----X----X----
JAR File
========
- JAR stands for Java ARchive.
- It is a zip file containing packages, sub-packages and compiled source
code (.class files).

WAR File
--------
- WAR stands for Web ARchive
- It is a zip file for Java web applications
- It contains
      - packages, sub-packages and compiled source code (.class files)
      - frontend files (.html, .css, .js, .png, .xml etc)

See Image: 01_jar_package.png

- Whenever we want to use a java library we have to download .jar files
- Remember : Every Java Library will contain .class executable files and
not .java source code files.

Modifiers in Java
=================
There are 2 types of modifiers in Java:
1.    Access Modifiers
2.    Non-access Modifiers

Access Modifiers
================
Access modifiers are used to control the visibility of a class, variable,
method or constructor.

There are 4 access modifiers in Java:
```

1. <default/package-private> - No Keyword
   Accessible only in the same package

2. public
   Accessible everywhere, but we have to import the classes

3. private
   Accessible only within the same class

4. protected
   Accessible within the same package and for all child classes (even if they are in other package)

Image : 02_access_modifiers.png

Example Setup: 03_access_modifiers.png

Create 3 packages: p1, p2, p3 in src folder and paste files accordingly

Non Access Modifiers
====================
- Non-access modifiers do not change the accessibility of classes, variables or methods, but they do provide them special properties.
- There are a total of 8 non-access modifiers in Java:

1. final - Prevents modification of classes, methods, or variables.
2. static - Belongs to the class, not instances.
3. abstract - Used for incomplete classes or methods to be defined by subclasses.
4. native - Links Java with platform-specific native code (e.g., C/C++).
5. strictfp - Enforces strict floating-point calculation rules for portability.
6. volatile - Ensures the variable is always read from memory, not cache.
7. transient - Prevents serialization of certain class fields.
8. synchronized - Controls access to methods or blocks by multiple threads.

-----X-----X-----X-----X-----