Java Packages and JAR Files - Detailed Notes

1. Packages

Definitions

1. A **Package** is a collection of related classes and interfaces grouped together as a single unit.
2. From a file system perspective, a **Package** is represented as a folder that contains the `.class` files corresponding to the related classes and interfaces.

Advantages of using Packages

Using packages provides several benefits for organizing and managing Java code:

Advantage	Description	Reasoning
Modularization	Packages group related classes and interfaces for a particular purpose, providing modularity.	Breaks down large applications into smaller, manageable units.
Abstraction	Classes/interfaces inside a package are not directly exposed outside, providing abstraction.	Hides internal implementation details from external access.
Security	By default, packages hide internal declarations, protecting them from external access.	Controls access using package-private access, protecting internal components.
Reusability	A package can be created once and used multiple times as needed.	Allows code units to be easily reused across projects and applications.
Shareability	Once created, a package can be shared to multiple Java programs or with other developers.	Facilitates collaboration and code sharing.

Export to Sheets

Types of Packages

There are two main types of packages in Java:

1. **Predefined Packages:** These are part of the Java Standard Library.
2. **User Defined Packages:** These are created by developers.

Predefined Packages

These packages are defined by the Java language creators and are bundled with the Java software.

Package	Purpose	Examples of Classes/Interfaces
`java.lang`	Fundamental or basic classes/interfaces. Automatically imported.	`Object`, `System`, `String`, `Number`, `Integer`, `Comparable`.
`java.io`	Input and output operations.	`InputStream`, `OutputStream`, `BufferedReader`, `FileWriter`, `Serializable`.
`java.util`	Data Structures (Collections Framework) and utility functions.	`List`, `ArrayList`, `Set`, `HashMap`, `Date`, `Calendar`, `Random`.
`java.awt`	GUI applications (Abstract Window Toolkit).	`Frame`, `TextField`, `Label`, `Button`, `Color`, `Font`.
`javax.swing`	More efficient and advanced GUI components (Swing).	`JFrame`, `JPanel`, `JButton`, `JTextField`, `JTextArea`.
`java.net`	Network programming (Socket programming).	`Socket`, `ServerSocket`, `URL`, `URLConnection`, `InetAddress`.
`java.rmi`	Distributed applications (Remote Method Invocation).	`RemoteException`, `Remote`, `Naming`, `UnicastRemoteObject`.
`java.sql`	Java Database Connectivity (JDBC) applications.	`Driver`, `DriverManager`, `Connection`, `Statement`, `ResultSet`, `PreparedStatement`.
`java.text`	Internationalization and formatting (numbers, dates, messages).	`NumberFormat`, `DateFormat`, `SimpleDateFormat`.

Export to Sheets

User Defined Packages

These packages are defined by developers to structure their specific application code.

To define a user-defined package, use the `package` keyword followed by the desired package name:

Java
```
package packageName;
```

- `packageName` can be a single identifier or a hierarchical name (parent.child.subchild).
  - Examples: `package p;`, `package p1.p2.p3;`
  - (Reasoning: Hierarchical names correspond to nested folders on the file system, helping organize larger projects.)

Conditions for `package` declaration:

1. The `package` declaration statement **must be the first statement** in the Java file (ignoring comments and whitespace).
2. The package name should be unique (globally unique is the convention).

Q) Is it possible to provide more than one package declaration statement in a single Java file? —---------- Ans: —---- No, it is not possible. The

`package` statement must be the *first* statement in the file. You cannot have a second first statement.

EX: abc.java

Java
```
package p1; // ------> Valid (This is the first statement)
// package p2; // ------> Invalid (Cannot be second)
// package p3; // ------> Invalid
```

Package Naming Convention: Java recommends reversing the company domain name for uniqueness, followed by project/module names.

- Example Domain: `durgasoft.com`
- Reversed Domain: `com.durgasoft`

Full EX:

Java
```
package com.durgasoft.icici.transactions.deposit;
```

- `com.durgasoft`: Company domain name in reverse, ensuring a unique base.
- `icici`: Client name or Project name (further organization within the company's domain).
- `transactions`: Module name.
- `deposit`: Sub-module name. (Reasoning: This convention, combined with the hierarchy, creates globally unique package names, preventing conflicts when integrating code from different sources.)

2. Importing Packages

To use classes and interfaces from a package, you must make them available using import statements. Import statements come after the package declaration and before class definitions.

Import Statements Syntax:

Syntax	Purpose	Example	Reasoning
`import packageName.*;`	Imports all public members (classes, interfaces, etc.) from package.	`import java.io.*;`	Convenient, but imports potentially unused classes.
`import packageName.Member;`	Imports only the Specified member (class, interface) from package.	`import java.util.ArrayList;`	More explicit, improves readability.

Export to Sheets

Note: It is always suggestible to import individual classes/interfaces instead of using `*` for better readability.

Example Code with Wildcard Imports:

Java
```
// Test.java
import java.io.*;
import java.util.*;
import java.sql.*;
public class Test{
        public static void main(String[] args){
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

                List list = new ArrayList();

                // Example using imported class (static method)
                // Connection con =
DriverManager.getConnection("--","--","--");
        }
}
```

Example Code with Individual Imports:

Java
```
// Test.java
import java.io.BufferedReader;      // Explicitly import
import java.io.InputStreamReader; // Also needed explicitly
import java.util.List;              // Explicitly import
import java.util.ArrayList; // Explicitly import
import java.sql.Connection; // Explicitly import
import java.sql.DriverManager; // Explicitly import
public class Test{
        public static void main(String[] args){
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

                List list = new ArrayList();

                // Example using imported class (static method)
                // Connection con =
DriverManager.getConnection("--","--","--");
        }
}
```

Note: You are able to provide at most one package declaration statement, but more than one import statement is allowed. EX: abc.java

Java
```
package p1; // --------> Valid
// package p2; // --------> Invalid
import java.io.*; // ---> Valid
import java.util.*; // ---> Valid
import java.sql.*; // ---> Valid
```

3. Using Classes Without Imports (Fully Qualified Names)

Q)Is it possible to use classes and interfaces of a particular package in the present java file without importing the respective package?

—---------- Ans: —--- Yes, it is possible, but we have to use the **fully qualified names** of the classes and interfaces each time they are referenced.

Note: Writing classes and interfaces names along with their respective package names is called a Fully qualified name. EX:

- `java.io.BufferedReader`
- `java.util.ArrayList`
- `java.sql.Connection`

Example code with import statement:

Java
```
import java.io.BufferedReader; // Import statement used
// import java.io.InputStreamReader; // Often also needed if not using
wildcard
BufferedReader br = new BufferedReader(new InputStreamReader( System.in));
```

Example code without import statement (using Fully Qualified Names):

Java
```
// No import java.io...
java.io.BufferedReader br = new java.io.BufferedReader(new
java.io.InputStreamReader( System.in));
// Note: Every reference to BufferedReader and InputStreamReader needs the
package prefix
```

Note: It is always suggestible to use import statements instead of Fully qualified names for readability.

Q)In which situations we must use fully qualified names to the classes and interfaces over the import statements in java applications? —---------- Ans: —---- If we have more than one class with the same simple name in more than one package, then we **must** use fully qualified names to specify which class is intended to avoid ambiguity.

(Reasoning: Using the simple name becomes ambiguous if the same class name exists in multiple imported packages. Fully qualified names remove this ambiguity.)

Example: Ambiguity with `java.util.Date` and `java.sql.Date`

Java Program with conflicting imports (ambiguous):

Java
```
// Test.java
import java.util.*; // Imports java.util.Date
import java.sql.*;  // Imports java.sql.Date
public class Test{
        public static void main(String[] args){
                Date sysDate = new Date(); // Ambiguous reference
                Date dbDate = new Date();  // Ambiguous reference
        }
}
```

```
// Status: Confusion, compilation fails.
```

Java Program resolving ambiguity using Fully Qualified Names:

Java

```
// Test.java
// No imports that cause Date ambiguity
public class Test{
        public static void main(String[] args){
                java.util.Date sysDate = new java.util.Date(); // Clearly
java.util.Date
                // Original code had parseDate - correcting to common
practice
                // java.sql.Date dbDate = java.sql.Date.parseDate("12-12-
2024");;
                java.sql.Date dbDate = java.sql.Date.valueOf("2024-12-12");
// Clearly java.sql.Date
        }
}
// Status: No Confusion, Compilation Successful.
```

(Reasoning: By using the fully qualified name, you remove the ambiguity, telling the compiler exactly which class you intend to instantiate or reference.)

4. Classpath

The "classpath" environment variable or command-line option tells the compiler (`javac`) and the JVM (`java`) where to find user-defined packages (directories or JAR files) and classes needed for compilation and execution.

- When compiling (`javac`), the classpath is used to find any classes referenced by the code being compiled.
- When running (`java`), the classpath is used to find the main class and any other classes needed at runtime.

The classpath is a list of locations (directories or JAR files). The Java tools search these locations for the root of package hierarchies (like the `com` folder) or for un-packaged classes.

Example 1: Using a package compiled to a separate directory

Scenario:

- `Employee.java` (package `com.durgasoft.emp`) in `E:\abc`.
- Compile `Employee.java` and place `.class` in `C:\abc`. This creates `C:\abc\com\durgasoft\emp\Employee.class`.
- `Test.java` (uses `Employee`) in `D:\java830\packages\app01`.

Command Line Steps:

1. Compile Employee.java:

   Bash

```
E:\abc>javac -d C:\abc Employee.java
```

(Reasoning: `-d C:\abc` sets the destination directory. Package subdirectories are created automatically.)

2. Navigate to Test.java directory:

   Bash

   ```
   E:\abc>d:
   D:\>cd java830\packages\app01
   D:\java830\packages\app01>
   ```

3. Attempt to compile Test.java (fails):

   Bash

   ```
   D:\java830\packages\app01>javac Test.java
   ```

   Output:

   ```
   Test.java:1: error: package com.durgasoft.emp does not exist
   import com.durgasoft.emp.*;
   ^
   ... (more errors about Employee not found)
   3 errors
   ```

   (Reasoning: Compiler doesn't know where to find the package.)

4. Set classpath to include the base directory of the package:

   Bash

   ```
   D:\java830\packages\app01>set classpath=C:\abc;
   ```

   (Reasoning: Tells Java tools to look inside `C:\abc` for packages like `com`. Semicolon separates paths.)

5. Compile Test.java (succeeds):

   Bash

   ```
   D:\java830\packages\app01>javac Test.java
   ```

   (Reasoning: Compiler finds the package structure starting from `C:\abc`.)

6. Attempt to run Test.class (fails):

   Bash

   ```
   D:\java830\packages\app01>java Test
   ```

Output:

```
Error: Could not find or load main class Test
Caused by: java.lang.ClassNotFoundException: Test
```

(Reasoning: JVM needs to find `Test.class` itself, which is in the current directory not in the classpath `C:\abc`.)

7. Set classpath to include package location AND current directory:

   Bash

   ```
   D:\java830\packages\app01>set classpath=C:\abc;.;
   ```

   (Reasoning: `.;` adds the current directory to the search path.)

8. Run Test.class (succeeds):

   Bash

   ```
   D:\java830\packages\app01>java Test
   ```

   Output:

   ```
   Employee Details
   ----------------------
   Employee Number     : 111
   Employee Name       : Durga
   Employee Salary     : 50000.0
   Employee Address    : Hyd
   ```

Example 2: Using multiple packages in different target directories

Scenario:

- `Student.java` (package `com.durgasoft.std`) from `E:\abc`, compiled to `C:\xyz` (`C:\xyz\com\durgasoft\std\Student.class`).
- `Customer.java` (package `com.durgasoft.cust`) from `C:\abc`, compiled to `E:\xyz` (`E:\xyz\com\durgasoft\cust\Customer.class`).
- `Test.java` (uses `Student` and `Customer`) from `D:\java830\packages\app02`.

Command Line Steps:

1. Compile Student.java:

   Bash

   ```
   E:\abc>javac -d C:\xyz Student.java
   ```

2. Compile Customer.java:

   Bash

```
C:\abc>javac -d E:\xyz Customer.java
```

3. Navigate to Test.java directory:

   Bash

   ```
   C:\abc>D:
   D:\>cd java830\packages\app02
   ```

4. Set classpath to include both package base directories:

   Bash

   ```
   D:\java830\packages\app02>set classpath=C:\xyz;E:\xyz;
   ```

   (Reasoning: Compiler needs to find packages starting in both directories.)

5. Compile Test.java (succeeds):

   Bash

   ```
   D:\java830\packages\app02>javac Test.java
   ```

6. Set classpath to include both package directories and current directory for running:

   Bash

   ```
   D:\java830\packages\app02>set classpath=C:\xyz;E:\xyz;.;
   ```

7. Run Test.class (succeeds):

   Bash

   ```
   D:\java830\packages\app02>java Test
   ```

   Output:

   ```
   Student Details
   -----------------------
   Student Id      : S-111
   Student Name    : Durga
   Student Address : Hyd
   Customer Details
   -----------------------
   Customer Id      : C-111
   Customer Name    : BBB
   Customer Address : Pune
   ```

Note: “classpath” makes packages available to compiler and JVM. Import statements make classes from available packages usable by simple name in source code.

5. JAR Files in Java

JAR: Java Archive. It is the collection of packages, .class files, .java files, and resources in a single file.

Purpose: JAR files simplify distribution (transferring projects), uploading/downloading software, and packaging applications.

Creating JAR Files

Command:

Bash
```
jar -cvf fileName.jar files_and_directories_to_include
```

JAR Creation Options:

Option	Meaning
`-c`	**c**reate a new archive
`-v`	**v**erbose output
`-f`	Specify archive **f**ile name

Export to Sheets

EX: `jar -cvf logo.jar *` (Creates logo.jar from current directory contents) (Reasoning: Packages contents into a single file for easier management.)

Extracting JAR Files

Command:

Bash
```
jar -xvf filename.jar
```

JAR Extraction Options:

Option	Meaning
`-x`	E**x**tract from archive
`-v`	**v**erbose output
`-f`	Specify archive **f**ile name

Export to Sheets

Accessing JAR File Content

Set the “classpath” environment variable to include the path to the JAR file.

Bash
```
set classpath=path/to/the/library.jar;.;
```

(Reasoning: Including the JAR in the classpath tells Java tools to look *inside* the JAR file for classes and package structures.)

Example 3: Creating a JAR and using it in another application

Scenario:

- `Account.java` (package `com.durgasoft.icici.accounts`) from `E:\abc`, compiled to `C:\abc`.
- Prepare jar file `icici_accounts.jar` from contents of `C:\abc`.
- `Test.java` (uses `Account`) from `D:\java830\packages\app03`.

Command Line Steps:

1. Compile Account.java:

   Bash

   ```
   E:\abc>javac -d C:\abc Account.java
   ```

2. Navigate to C:\abc and create JAR:

   Bash

   ```
   E:\abc>c:
   C:\>cd abc
   C:\abc>jar -cvf icici_accounts.jar *
   added manifest
   adding: com/(in = 0) (out= 0)(stored 0%)
   adding: com/durgasoft/(in = 0) (out= 0)(stored 0%)
   adding: com/durgasoft/icici/(in = 0) (out= 0)(stored 0%)
   adding: com/durgasoft/icici/accounts/(in = 0) (out= 0)(stored 0%)
   adding: com/durgasoft/icici/accounts/Account.class(in = 1447) (out=
   732)(deflated 49%)
   ```

3. (Optional) Move JAR to E:\xyz.
4. Navigate to Test.java directory:

   Bash

   ```
   C:\abc>d:
   D:\>cd java830\packages\app03
   ```

5. Attempt compile Test.java (fails):

   Bash

   ```
   D:\java830\packages\app03>javac Test.java
   Test.java:1: error: package com.durgasoft.icici.accounts does not
   exist
   ... (errors)
   ```

   (Reasoning: Compiler cannot find the package without the JAR in the classpath.)

6. Set classpath to include the JAR (assuming it's at E:\xyz):

Bash

```
D:\java830\packages\app03>set classpath=E:\xyz\icici_accounts.jar;
```

7. Compile Test.java (succeeds):

   Bash

   ```
   D:\java830\packages\app03>javac Test.java
   ```

   (Reasoning: Compiler finds `Account.class` inside the JAR.)

8. Attempt run Test.class (fails):

   Bash

   ```
   D:\java830\packages\app03>java Test
   Error: Could not find or load main class Test
   Caused by: java.lang.ClassNotFoundException: Test
   ```

   (Reasoning: JVM needs to find `Test.class` in the current directory, which is not in the classpath.)

9. Set classpath to include JAR AND current directory:

   Bash

   ```
   D:\java830\packages\app03>set classpath=E:\xyz\icici_accounts.jar;.;
   ```

10. Run Test.class (succeeds):

    Bash

    ```
    D:\java830\packages\app03>java Test
    Account Details
    --------------------
    Account Number          : abc123
    Account Holder Name     : Durga
    Account Type            : Savings
    Account Balance         : 50000
    ```

    (Reasoning: JVM finds `Test.class` in the current directory and `Account.class` inside the JAR.)

6. MANIFEST File

- MANIFEST is a file created by the `jar` command under the `META-INF` folder within a JAR file (`META-INF\MANIFEST.MF`).
- Purpose: Provides metadata about the JAR in the form of key-value pairs (attributes).

7. Executable Jar Files

- Definition: A jar file that contains a main class and a `Main-Class` entry in its MANIFEST.MF file, allowing it to be executed directly using `java -jar`.

Steps to prepare executable jar files:

1. Prepare and compile Java application with a main class:

   Java

```
// LogoFrame.java (Example GUI class)
import java.awt.*;
import java.awt.event.*;
public class LogoFrame extends Frame {
    public LogoFrame(){
        this.setVisible(true); this.setSize(900, 400);
this.setTitle("LOGO Frame");
        this.setBackground(Color.red);
        this.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent we)
{ System.exit(0); }
        });
    }
    public void paint(Graphics g){
        Font font = new Font("arial", Font.BOLD, 45);
g.setFont(font);
        this.setForeground(Color.white);
        g.drawString("DURGA SOFTWARE SOLUTIONS", 100,200);
    }
}
// Test.java (Main class)
public class Test {
    public static void main(String[] args) {
        LogoFrame logo = new LogoFrame(); // Starts the application
    }
}
```

   Compile:

   Bash

```
D:\java830\packages\app04>javac *.java
```

   (Reasoning: Compiles source files to .class files needed for the JAR.)

2. Provide Main Class attribute in a text file: Create a text file (e.g., `manifest_attr.txt`) containing `Main-Class: YourMainClass` followed by a newline.

```
3. Main-Class: Test
```

   (Reasoning: This file provides the specific attribute needed for the manifest.)

4. Create Jar file including the custom Manifest data: Use the `jar` command with the `-m` option.

   Bash

```
D:\java830\packages\app04>jar -cvfm logo.jar manifest_attr.txt
*.class
added manifest
adding: manifest_attr.txt(in = 18) (out= 20)(deflated -11%)
adding: LogoFrame$1.class(in = 505) (out= 344)(deflated 31%)
adding: LogoFrame.class(in = 993) (out= 644)(deflated 35%)
adding: Test.class(in = 285) (out= 223)(deflated 21%)
```

JAR Creation (Executable) Options: | Option | Meaning | | :---------- | :---------- | | `-c` | **c**reate new archive | | `-v` | **v**erbose output | | `-f` | Specify archive **f**ile name | | `-m manifest`| Include manifest info from `manifest` file | (Reasoning: `-m` adds the `Main-Class` entry from `manifest_attr.txt` to the JAR's manifest.)

5. Execute the JAR file:

   Bash

   ```
   D:\java830\packages\app04>java -jar logo.jar
   ```

How the JVM executes an Executable JAR (`java -jar`):

1. JVM recognizes the -jar option and the jar file name.
2. JVM searches for and opens the jar file.
3. JVM looks for the META-INF folder and MANIFEST.MF file inside the JAR.
4. JVM searches for the Main-Class attribute value in MANIFEST.MF.
5. JVM loads the class specified by Main-Class (e.g., `Test`) and executes its main() method. (Reasoning: The JAR's internal classes are automatically on the classpath for this execution.)

Simplifying Execution with a Batch File

Create a simple batch file (`RunApp.bat`) with the execution command:

Code snippet
```
java -jar logo.jar
```

(Reasoning: Double-clicking the batch file runs the command, launching the Java application easily.) Copy the `.jar` and `.bat` file together for simple distribution.