Java Packages

What is a Package in Java?

A **package** in Java is a mechanism for organizing Java classes and interfaces into namespaces, providing access protection, and avoiding naming conflicts. It's a way to group related classes and interfaces into a directory structure.

Think of a package like a folder on your computer that contains similar files. In Java, packages help to:

- Organize your code.
- Control access to classes and methods.
- Avoid class name conflicts when using external libraries.

Types of Packages in Java

1. **Built-in Packages**: These are provided by Java and part of the Java API.
    - Examples: `java.util`, `java.lang`, `java.io`, `java.net`, etc.
2. **User-defined Packages**: These are packages created by users to group their own related classes and interfaces.
    - Example: `com.mycompany.project`, `org.example.utilities`.

Creating a Package

To create a package, use the `package` keyword at the top of your Java source file. The package statement should be the first line in your code (except comments).

Syntax:

```
package package_name;
```

Example:

Let's create a package `com.example.myapp` and a class inside it.

```
// File: com/example/myapp/MyClass.java
package com.example.myapp;

public class MyClass {
    public void sayHello() {
        System.out.println("Hello from MyClass in com.example.myapp package");
    }
}
```

The folder structure on your system should mirror the package name:

- `com/example/myapp/MyClass.java`

In this structure:

- **com**: is the top-level package.
- **example**: is a sub-package inside `com`.
- **myapp**: is a sub-package inside `example`.

Using a Package

To use a class from a package, you must import it into your file using the `import` keyword. If the class is in the same package, there's no need to import it.

Syntax for Importing a Package:

1. Import a specific class:

    ```
    import package_name.ClassName;
    ```
2. Import all classes in a package:

    ```
    import package_name.*;
    ```

Example:

Let's create a program that uses the `MyClass` we created earlier from the package `com.example.myapp`.

```
// File: TestPackage.java
```

```
import com.example.myapp.MyClass;

public class TestPackage {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.sayHello();
    }
}
```

In this example:

- We import `MyClass` from the `com.example.myapp` package.
- We create an instance of `MyClass` and call its `sayHello()` method.

If we want to import all classes in the `com.example.myapp` package:

```
import com.example.myapp.*;
```

Built-in Packages in Java

Java provides many built-in packages that contain classes for different functionalities. Here are some commonly used packages:

1. **java.lang**: Contains fundamental classes like `String`, `Math`, `Integer`, `Thread`, etc. It is automatically imported into every Java program.
   - Example: `String`, `System`, `Math`.
2. **java.util**: Contains utility classes, like data structures (e.g., `ArrayList`, `HashMap`), dates, and collections.
   - Example: `ArrayList`, `HashMap`.
3. **java.io**: Contains classes for input and output operations, such as file handling.
   - Example: `File`, `InputStream`, `OutputStream`.
4. **java.net**: Provides classes for networking, like sockets and URLs.
   - Example: `Socket`, `URL`.
5. **java.sql**: Provides classes for database connectivity using JDBC (Java Database Connectivity).
   - Example: `Connection`, `ResultSet`.

Example: Using java.util.ArrayList

```
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Packages");
        System.out.println(list);
    }
}
```

In this example:

- We import `ArrayList` from `java.util`.
- We create a list of strings and print it.

Access Modifiers and Packages

The access level of classes and methods in Java depends on their **access modifiers**:

1. **Public**: The class, method, or variable is accessible from any other class or package.
2. **Private**: The method or variable is accessible only within its own class.
3. **Protected**: The method or variable is accessible within its own package and by subclasses.
4. **Default (Package-private)**: If no modifier is specified, the method or variable is accessible only within its own package.

Example:

```
// File: com/example/Person.java
package com.example;

public class Person {
    public String name;    // public - accessible everywhere
    private int age;       // private - accessible only within this class
    protected String address;  // protected - accessible within the package and subclasses
}
```
In another class:

```
// File: com/example/Employee.java
package com.example;

public class Employee extends Person {
    public void showDetails() {
        System.out.println("Name: " + name);    // Accessible (public)
        // System.out.println("Age: " + age);  // Not accessible (private)
        System.out.println("Address: " + address); // Accessible (protected)
    }
}
```

Sub-Packages in Java

Java allows for the creation of **sub-packages** within a package. A sub-package is simply a package inside another package. The naming convention for sub-packages is to separate them by periods (`.`).

Example:
```
package com.example.utils;

public class Utility {
    public void performTask() {
        System.out.println("Performing a utility task.");
    }
}
```
Folder structure:

- `com/example/utils/Utility.java`

In this example, `com.example.utils` is a sub-package of `com.example`.

Using the Classpath

The **classpath** in Java is the parameter that tells the Java runtime and compiler where to look for user-defined classes and packages. When compiling or running a program, you can specify the classpath using the `-cp` or `-classpath` option.

Compiling with Classpath:
```
javac -cp . com/example/myapp/MyClass.java
```
Running with Classpath:
```
java -cp . com.example.myapp.MyClass
```
In this example, `.` (dot) refers to the current directory as the classpath.

Package Sealing

Java provides a feature called **package sealing**, which ensures that all classes within a package come from the same source (like a JAR file). This prevents the addition of classes from a different source to the same package.

Example:
You can seal a package by adding this entry to the JAR file's manifest (`META-INF/MANIFEST.MF`):

```
Sealed: true
```

Real-World Example: Creating a User-defined Package

Let's create a user-defined package that represents a **library management system**.

1. **Create a package `com.library`:**

   ```
   package com.library;

   public class Book {
       private String title;
       private String author;
   ```

```
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public void display() {
        System.out.println("Title: " + title + ", Author: " + author);
    }
}
```

2. **Create another class `Member` in the same package**:

```
package com.library;

public class Member {
    private String name;

    public Member(String name) {
        this.name = name;
    }

    public void showDetails() {
        System.out.println("Member Name: " + name);
    }
}
```

3. **Create a class to use `Book` and `Member`**:

```
import com.library.Book;
import com.library.Member;

public class LibraryTest {
    public static void main(String[] args) {
        Book book = new Book("Effective Java", "Joshua Bloch");
        Member member = new Member("Alice");

        book.display();
        member.showDetails();
    }
}
```

- **Folder Structure:**

```
com/library/Book.java
com/library/Member.java
LibraryTest.java
```

In this example, the classes `Book` and `Member` are part of the `com.library` package, and we import and use them in the `LibraryTest` class.

Advantages of Using Packages

1. **Namespace Management:** Packages prevent class name conflicts. For example, `com.bank.Account` and `com.library.Account` can coexist.
2. **Modularity:** Packages help organize large projects by grouping related classes and interfaces.
3. **Reusability:** Classes in packages can be reused across projects.
4. **Access Control:** Packages provide a mechanism for controlling the visibility of classes, methods, and variables using access modifiers.
5. **Maintenance:** Packages make it easier to maintain and navigate large codebases.