SILVER OAK UNIVERSITY
EDUCATION TO INNOVATION

Course Code : 3040243201
Course Name: Java Programming
Unit No: 3
Unit Name: Packages & Interfaces & Exception Handling

SEMESTER: 3

PREPARED BY: Dr. Nikunj Raval

- **Package** in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:
- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

How packages work?

- Packages help organize your Java code and prevent naming conflicts. If you're looking to master package creation and management in large Java applications, the **Java Programming Course** provides comprehensive lessons and hands-on exercises
- Package names and directory structure are closely related. For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff*and *cse* such that *cse* is present in *staff* and *staff* is present inside *college*. Also, the directory *college* is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.
- **Package naming conventions :** Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.
- **Adding a class to a Package :** We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.
- **Subpackages:** Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Example : import java.util.*;

util is a subpackage created inside **java** package.

Types of packages:

- **Built-in Packages**
- These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

1. **java.lang:** Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
2. **java.io:** Contains classes for supporting input / output operations.
3. **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4. **java.applet:** Contains classes for creating Applets.
5. **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc). 6)
6. **java.net:** Contain classes for supporting networking operations.

User-defined packages: These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names.**

- In Java, packaging classes is an essential practice for organizing your code, managing namespaces, and preventing naming conflicts. Here's a concise overview of how to package your classes effectively:
- **1. Understanding Packages**
- A package in Java is a namespace that organizes a set of related classes and interfaces. It helps in avoiding name clashes and can also control access with protected and default access levels.

2. Creating a Package

To create a package, you use the package keyword at the top of your Java source file. The package declaration should be the first line in the file, followed by any import statements.

Example:

java

```
package com.example.myapp;
public class MyClass {
// Class content here
}
```

3. Compiling Classes

When you compile your classes, you need to maintain the directory structure that reflects the package name. For example, the package com.example.myapp should be stored in a directory structure like com/example/myapp/MyClass.java.

4. Using Packages

To use classes from a package, you need to import them in your Java files.

Example:

java

```
import com.example.myapp.MyClass;
public class Test {
public static void main(String[] args) {
MyClass myClass = new MyClass();
 // Use myClass
}
}
```

5. Default Package

- If you don't specify a package, your class is part of the default package. However, it's a good practice to avoid using the default package in larger applications.

6. Access Modifiers and Packages

- **Public**: The class can be accessed from any other class.
- **Protected**: The class can be accessed by classes in the same package and subclasses.
- **Default** (no modifier): The class can be accessed only by classes in the same package.

7. Best Practices

- **Naming Conventions**: Use reverse domain names for packages (e.g., com.yourcompany.project).
- **Organize Logically**: Group related classes into the same package.
- **Documentation**: Use Javadoc to document your classes and packages for clarity.

8. Jar Files

jar cf myapp.jar -C path/to/classes .

1. Using Classes from a Package

To use a class from a package, you typically need to import it. This can be done with the import statement. For example, if you have a class MyClass in the package com.example.myapp, you would import it as follows:

java

```
import com.example.myapp.MyClass;
public class Test {
public static void main(String[] args) {
MyClass myClass = new MyClass();
// Use myClass methods
}
}
```

2. Understanding CLASSPATH

- The CLASSPATH is an environment variable that tells the Java Virtual Machine (JVM) and Java compiler where to look for user-defined classes and packages. If your classes or libraries aren't in the default Java installation directories, you need to specify their locations in the CLASSPATH.

Setting the CLASSPATH

1. **Temporarily in the Command Line**: You can set the CLASSPATH for a single session in your terminal or command prompt.

- set CLASSPATH=C:\path\to\classes;C:\path\to\jarfile.jar;.

3. Compiling and Running with CLASSPATH

- When you compile your program, the CLASSPATH helps the compiler locate the classes you're importing. For example, if you have your classes compiled into a directory structure reflecting their packages, you would compile your Java files like this:
- javac -cp . Test.java
- java -cp . Test
- **4. Using JAR Files**
- export CLASSPATH=/path/to/mylibrary.jar:.
- **5. Classpath Wildcards**
- set CLASSPATH=C:\path\to\libs*;.

- Exception handling in Java is a powerful mechanism that helps you manage errors and other exceptional conditions that may arise during program execution
- **1. The Idea Behind Exceptions**
- Exceptions are unexpected events that occur during the execution of a program, disrupting the normal flow of instructions. They can be caused by various issues, such as:
- Invalid user input
- File not found
- Network errors
- Division by zero
- w of exception handling in Java

2. Types of Exceptions

Java has two main categories of exceptions:

a. Checked Exceptions

- **Definition**: These are exceptions that must be either caught or declared in the method signature. They are checked at compile time.
- **Examples**: IOException, SQLException, FileNotFoundException.

b. Unchecked Exceptions

- **Definition**: These exceptions do not need to be declared or caught. They are checked at runtime.
- **Examples**: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.

3. Dealing with Exceptions

In Java, you deal with exceptions using try, catch, finally, and throw statements.

a. Try and Catch Blocks

You wrap the code that might throw an exception in a try block and handle the exception in a catch block.

```
try {
    // Code that may throw an exception
    int result = 10 / 0; // This will throw ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
}
```

b. Finally Block

The finally block is used for cleanup actions that must excute regardless of whether an exception was thrown or not .

```
try {
    // Code that may throw an exception
} catch (Exception e) {
    // Handle exception
} finally {
    // Code that runs regardless of exception
    System.out.println("This will always execute.");
}
```

- c. **Throwing Exceptions**
- You can explicitly throw an exception using the throw keyword
- if (age < 18) {
- throw new IllegalArgumentException("Age must be at least 18.");
- }
- **4. Exception Objects**
- When an exception occurs, an instance of an exception class (an exception object) is created. This object contains:
- **Error message**: Describes the nature of the error.
- **Stack trace**: Information about the method call stack at the time the exception occurred, useful for debugging.

```
try {
    // Code that may throw an exception
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
    e.printStackTrace(); // Print stack trace for debugging
}
```

5. **Defining Your Own Exceptions**

You can create custom exception classes by extending the Exception class (for checked exceptions) or RuntimeException (for unchecked exceptions).

Example of a Custom Checked Exception:

```
class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);}}
```

Using the Custom Exception:

```
public void myMethod() throws MyCustomException {
    // Some code
    if (/* some condition */) {
        throw new MyCustomException("This is my custom exception.");
    }
}
```