

JAVA PROGRAMMING

Chapter 3

Interfaces and Packages

3.1 Interface: Multiple Inheritance

3.2 Packages: Putting Classes Together

Detailed Notes for Engineering Students

Theory + Programs + Tables + 15 MCQs with Answer Key + 10 Broad Questions

PolyNotesHub

Contents

3.1 Interface: Multiple Inheritance	3
3.1.1 Introduction	3
3.1.2 Defining Interfaces	4
3.1.3 Extending Interfaces	4
3.1.4 Implementing Interfaces	6
3.1.5 Accessing Interface Variables	8
3.2 Packages: Putting Classes Together	10
3.2.1 Introduction	10
3.2.2 System Packages	10
3.2.3 Using System Packages	11
3.2.4 Naming Conventions	12
3.2.5 Creating a Package	12
3.2.6 Accessing a Package	13
3.2.7 Using a Package	14
3.2.8 Adding a Class to a Package	15
Multiple Choice Questions (15)	17
Answer Key	19
Broad Questions (10)	20

3.1 Interface: Multiple Inheritance

3.1.1 Introduction

Inheritance is the mechanism by which a class acquires the properties (variables) and behaviour (methods) of another class. Java supports **single**, **multilevel** and **hierarchical** inheritance directly through classes using the keyword `extends`. However, Java **does not allow a class to extend more than one class**. A statement such as `class C extends A, B` is a compile-time error.

The reason is the **ambiguity problem** (also called the *diamond problem*): if two parent classes contain a method with the same name but different bodies, the compiler cannot decide which version the child should inherit. To keep the language simple and safe, Java's designers removed multiple inheritance of classes. But real-world designs often need a class to follow more than one 'contract'. Java solves this need with **interfaces**.

An **interface** is a reference type, similar to a class, that is a collection of **abstract methods** and **constants**. It describes *what* a class must do, but not *how* it is done. A class can implement any number of interfaces, and this is how Java achieves the effect of **multiple inheritance**.

Types of inheritance in Java

Type	Description	Supported using
Single	One subclass extends one superclass	Classes (extends)
Multilevel	A chain: C extends B, B extends A	Classes (extends)
Hierarchical	Several subclasses extend one superclass	Classes (extends)
Multiple	One class inherits from more than one parent	Interfaces only (implements)
Hybrid	Combination of the above types	Classes + interfaces

Table 3.1: Inheritance types and Java support

Difference between a class and an interface

Basis	Class	Interface
Keyword	<code>class</code>	<code>interface</code>
Instantiation	Objects can be created using <code>new</code>	Objects cannot be created
Methods	Concrete methods (with body)	Abstract methods (no body); default/static methods allowed from Java 8
Variables	Any type of variable (instance, static, final)	Only constants: implicitly public static final
Constructors	Can have constructors	Cannot have constructors
Inheritance	Extends only one class	A class can implement many interfaces; an interface can extend many interfaces
Access of members	private, default, protected, public	Members are public by default
Purpose	Defines state and behaviour of objects	Defines a contract/specification to be followed

3.1.2 Defining Interfaces

An interface is defined much like a class. The general form is:

General syntax

```
accessSpecifier interface InterfaceName
{
    type VARIABLE1 = value;        // constants
    type VARIABLE2 = value;
    returnType method1(parameter-list); // abstract methods
    returnType method2(parameter-list);
}
```

- **accessSpecifier** is either `public` or omitted. If omitted, the interface is available only inside its own package (package-private).
- **interface** is the keyword, followed by a valid identifier as the interface name (by convention, starting with a capital letter).
- **Variables** declared in an interface are automatically `public static final`, i.e. constants. They **must be initialised** at the point of declaration.
- **Methods** declared in an interface are automatically `public abstract`. They end with a semicolon and have no body.
- An interface cannot contain constructors, instance variables or instance initialisation blocks.
- The source file of a public interface must have the same name as the interface (`Area.java`).

Example 1: Defining an interface

```
interface Area
{
    final static float PI = 3.14F;        // constant
    float compute(float x, float y);      // abstract method
    void show();                          // abstract method
}
```

Modern features (Java 8 and later)

Feature	Introduced in	Details
default method	Java 8	Method with a body inside an interface, marked with the keyword <code>default</code> . Implementing classes may use or override it.
static method	Java 8	Utility method with a body, called as <code>InterfaceName.method()</code> . Not inherited by implementing classes.
private method	Java 9	Helper method used only by other methods of the same interface.

Table 3.3: Newer interface members

3.1.3 Extending Interfaces

Like classes, interfaces can also be extended. A new interface can inherit all the members of an existing interface using the keyword `extends`. The new interface is called the **sub-interface** and the existing one the **super-interface**. A class that implements the sub-interface must implement the abstract methods of **both** the sub-interface and the super-interface.

Syntax: single extension

```
interface Name2 extends Name1
{
    // body of Name2
}
```

Unlike classes, an interface **can extend several interfaces** at the same time, separated by commas. This is another way Java achieves multiple inheritance.

Syntax: multiple extension

```
interface Name3 extends Name1, Name2
{
    // body of Name3
}
```

Example 2: Interface extending two interfaces

```
interface ItemConstants
{
    int code = 1001;
    String name = "Fan";
}
interface ItemMethods
{
    void display();
}
interface Item extends ItemConstants, ItemMethods
{
    void price();
}
class Fan implements Item
{
    public void display() { System.out.println(code + " " + name); }
    public void price() { System.out.println("Price: Rs. 1500"); }
}
public class Ext
{
    public static void main(String[] args)
    {
        Fan f = new Fan();
        f.display();
        f.price();
        System.out.println(Item.code);
    }
}
```

Output

```
1001 Fan
Price: Rs. 1500
1001
```

- An interface extends another interface using **extends**, never **implements**.
- An interface **cannot** extend a class.
- An interface **cannot implement** another interface (it has no method bodies to provide).
- The sub-interface inherits all constants and abstract methods of its super-interfaces.

3.1.4 Implementing Interfaces

Interfaces are used as 'super-classes' whose properties are inherited by classes. A class uses the keyword `implements` to accept the contract of an interface. The general form is:

Syntax

```
class ClassName extends SuperClass implements Interface1, Interface2
{
    // class body: must define ALL abstract methods
}
```

- If a class both extends a class and implements interfaces, the **extends clause comes first**, followed by **implements**.
- A class can implement **any number** of interfaces; names are separated by commas.
- The class **must provide a body for every abstract method** of the interface; otherwise the class must be declared **abstract**.
- Implemented methods must be declared **public**, because interface methods are public and a method cannot reduce visibility when overriding.
- The method signature (name, parameters, return type) must match the one declared in the interface.
- An interface **reference variable** can hold the object of any class that implements it. This gives **run-time polymorphism**.

Example 3: Implementing an interface in two classes

```
interface Area
{
    final static float PI = 3.14F;
    float compute(float x, float y);
    void show();
}
class Rectangle implements Area
{
    public float compute(float x, float y) { return x * y; }
    public void show() { System.out.println("Shape: Rectangle"); }
}
class Circle implements Area
{
    public float compute(float x, float y) { return PI * x * x; }
    public void show() { System.out.println("Shape: Circle"); }
}
public class InterfaceTest
{
    public static void main(String[] args)
    {
        Area area; // interface reference
        area = new Rectangle();
        area.show();
        System.out.println("Area of Rectangle = " + area.compute(10, 20));
        area = new Circle();
        area.show();
        System.out.println("Area of Circle = " + area.compute(2, 0));
    }
}
```

Output

```
Shape: Rectangle
Area of Rectangle = 200.0
Shape: Circle
Area of Circle = 12.56
```

Multiple inheritance through interfaces (hybrid example)

In the program below, class **Results** inherits from class **Test** (which itself extends **Student**) and also implements the interface **Sports**. Thus **Results** receives features from more than one source, which is multiple (hybrid) inheritance.

Example 4: Student - Test - Sports - Results

```
class Student
{
    int rollNumber;
    void getNumber(int n) { rollNumber = n; }
    void putNumber() { System.out.println("Roll No : " + rollNumber); }
}
class Test extends Student
{
    float part1, part2;
    void getMarks(float m1, float m2) { part1 = m1; part2 = m2; }
    void putMarks()
    {
        System.out.println("Marks obtained");
        System.out.println("Part1 = " + part1);
        System.out.println("Part2 = " + part2);
    }
}
interface Sports
{
    float sportWt = 6.0F;
    void putWt();
}
class Results extends Test implements Sports
{
    float total;
    public void putWt() { System.out.println("Sports Wt = " + sportWt); }
    void display()
    {
        total = part1 + part2 + sportWt;
        putNumber();
        putMarks();
        putWt();
        System.out.println("Total score = " + total);
    }
}
public class Hybrid
{
    public static void main(String[] args)
    {
        Results student1 = new Results();
        student1.getNumber(1234);
        student1.getMarks(27.5F, 33.0F);
        student1.display();
    }
}
```

Output

```
Roll No : 1234
Marks obtained
Part1 = 27.5
Part2 = 33.0
Sports Wt = 6.0
Total score = 66.5
```

extends vs implements

Point	extends	implements
Used between	class-class and interface-interface	class-interface
Number allowed	Class: only one. Interface: many	Many interfaces
Method bodies	Inherited methods are already defined	Class must define all abstract methods
Order	Written first	Written after extends

Table 3.4: extends vs implements

3.1.5 Accessing Interface Variables

Interface variables are **constants**: they are implicitly `public`, `static` and `final`. Because they are static, they belong to the interface itself and no object is needed to access them. They are used to import shared constants into many classes, similar to `#define` constants in C.

- **Using the interface name:** `Area.PI`
- **Directly inside an implementing class:** simply write `PI`, because the class inherits the constant.
- **Using the class name or object of the implementing class:** `Circle.PI` is also valid.
- **Modification is not allowed.** Writing `Area.PI = 3.5F;` gives a compile-time error because the variable is `final`.
- The variable **must be initialised** when declared; a blank constant is an error.

Way of access	Example	Valid?
InterfaceName.VARIABLE	<code>Area.PI</code>	Yes
Inside implementing class	<code>PI</code>	Yes
ImplementingClass.VARIABLE	<code>Circle.PI</code>	Yes
Assigning a new value	<code>Area.PI = 3.5F;</code>	No (final)
Declaring without value	<code>float PI;</code>	No (must initialise)

Table 3.5: Accessing interface variables

Ambiguity when two interfaces have the same variable name

If a class implements two interfaces that contain a variable with the same name, using the bare name inside the class is ambiguous and the compiler reports an error (*reference to X is ambiguous*). The ambiguity is removed by qualifying the name with the interface name.

Example 5: Resolving ambiguity

```
interface A { int X = 10; }
interface B { int X = 20; }
class C implements A, B
{
    void show()
    {
        System.out.println("A.X = " + A.X);    // 10
        System.out.println("B.X = " + B.X);    // 20
        // System.out.println(X);    <-- compile-time error: ambiguous
    }
}
```

Default method conflict (Java 8+)

If a class implements two interfaces that both provide a `default` method with the same signature, the class **must override** that method. Inside it, a particular version can be called using `InterfaceName.super.method()`.

Points to remember: Interfaces

- An interface is a 100% abstract specification (before Java 8); it cannot be instantiated.
- Variables are public static final; methods are public abstract by default.
- A class implements interfaces; an interface extends interfaces.
- Multiple inheritance in Java is achieved only through interfaces.
- Implementing methods must be public and must match the interface signature.
- A class that does not implement all methods must be declared abstract.
- An interface with no members is called a **marker interface** (for example `Serializable`, `Cloneable`).

Common errors

Mistake	Reason / Result
<code>new Area();</code>	Interfaces cannot be instantiated: compile-time error
<code>void show() { ... }</code> in class (without public)	Weaker access privilege than interface method: compile-time error
<code>class C extends A, B</code> (A and B are classes)	Multiple class inheritance not allowed
<code>interface I implements J</code>	Interfaces use <code>extends</code> , not <code>implements</code>
Not defining all abstract methods	Class must be declared abstract or error occurs
<code>PI = 3.5F;</code> (inside implementing class)	Constant is final and cannot be changed

Table 3.6: Common mistakes with interfaces

3.2 Packages: Putting Classes Together

3.2.1 Introduction

A **package** is a group of related classes, interfaces and sub-packages bundled together under one name. A package works like a **folder** in a file system: just as related files are kept in one folder, related classes are kept in one package. Packages are the Java way of organising large programs into manageable parts.

Benefits of packages

- **Organisation:** related classes and interfaces are grouped, which makes programs easier to find and maintain.
- **Name-clash avoidance:** two classes with the same name can exist in different packages (`java.util.Date` and `java.sql.Date`).
- **Access protection:** the access modifiers protected and default work at package level to control visibility.
- **Reusability:** classes written once can be imported into any number of programs.
- **Hiding of classes:** classes not declared public stay hidden from outside the package.
- **Easy distribution:** a package can be bundled into a JAR file.

Classification of packages

Category	Description	Examples
Built-in (System) packages	Provided with the JDK as the Java API	<code>java.lang</code> , <code>java.util</code> , <code>java.io</code> , <code>java.awt</code>
User-defined packages	Created by the programmer	<code>package1</code> , <code>com.polynoteshub.util</code>

Table 3.7: Types of packages

3.2.2 System Packages

The Java system (API) packages are the large collection of ready-made classes and interfaces supplied with the JDK. They are organised in a hierarchy: the top-level names are `java` and `javax`, and below them are the individual packages. The dot (.) operator is used to separate the levels, for example `java.util.Scanner` means class `Scanner` in package `util` inside `java`.

Package	Contents / Purpose	Important classes
<code>java.lang</code>	Language support classes; imported automatically	<code>String</code> , <code>Math</code> , <code>Object</code> , <code>System</code> , <code>Integer</code> , <code>Thread</code> , <code>Exception</code>
<code>java.util</code>	Utility classes and collections framework	<code>Scanner</code> , <code>ArrayList</code> , <code>HashMap</code> , <code>Date</code> , <code>Random</code> , <code>Vector</code>
<code>java.io</code>	Input and output through streams and files	<code>File</code> , <code>FileReader</code> , <code>BufferedReader</code> , <code>InputStream</code> , <code>PrintWriter</code>
<code>java.net</code>	Networking support	<code>Socket</code> , <code>ServerSocket</code> , <code>URL</code>
<code>java.awt</code>	Abstract Window Toolkit: graphical user interface	<code>Frame</code> , <code>Button</code> , <code>Label</code> , <code>Graphics</code> , <code>Color</code>
<code>javax.swing</code>	Lightweight GUI components (extended GUI)	<code>JFrame</code> , <code>JButton</code> , <code>JLabel</code> , <code>JTextField</code>

Package	Contents / Purpose	Important classes
java.sql	Database access using JDBC	Connection, Statement, ResultSet
java.time	Date and time API (Java 8+)	LocalDate, LocalTime, Duration
java.text	Formatting of text, dates and numbers	SimpleDateFormat, DecimalFormat

3.2.3 Using System Packages

To use a class of a package in a program, the compiler must know where the class is located. This is done using the `import` statement, placed after the package statement (if any) and before the first class declaration. There are three ways:

Method	Syntax	Meaning
Import a single class	<code>import java.util.Scanner;</code>	Only Scanner becomes available by its simple name
Import all classes (on-demand)	<code>import java.util.*;</code>	All classes of java.util become available (not its sub-packages)
Fully qualified name	<code>java.util.Scanner sc = new java.util.Scanner(System.in);</code>	No import needed; full path is written every time

Table 3.9: Ways of using a system package

Example 6: Using classes of java.util

```
import java.util.Scanner;
import java.util.ArrayList;
public class UsePackage
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        ArrayList<String> names = new ArrayList<String>();
        System.out.print("Enter a name: ");
        names.add(sc.nextLine());
        System.out.println("List = " + names);
    }
}
```

- **java.lang** is imported automatically into every program; so String, Math and System can be used without any import statement.
- `import java.util.*;` does **not** import classes of sub-packages such as `java.util.concurrent`.
- Using the asterisk does not slow down the program at run time; it only tells the compiler where to search. For readability, importing individual classes is preferred.
- If two imported packages contain a class with the same name (for example Date), use the fully qualified name to avoid ambiguity.
- The import statement is not like `#include` in C; it does not copy code, it only introduces names.

3.2.4 Naming Conventions

Packages are named in a standard way so that names remain unique across the world. Sun/Oracle recommends that package names be written in **lowercase** letters and be based on the creator's **reversed Internet domain name**. For example, a company with the domain `polynotesHub.co.in` would name its packages `in.co.polynotesHub.util`. Package names are separated by dots; every dot corresponds to a folder level on disk.

Identifier	Convention	Example
Package	All lowercase; reverse domain name; dot separated	<code>com.mycompany.project</code> , <code>package1</code>
Class	Noun; first letter of each word capital (PascalCase)	<code>Student</code> , <code>ArrayList</code>
Interface	Capitalised like classes; often adjective or noun	<code>Runnable</code> , <code>Area</code> , <code>Sports</code>
Method	Verb; camelCase	<code>getMarks()</code> , <code>displayA()</code>
Variable	camelCase; meaningful names	<code>rollNumber</code> , <code>total</code>
Constant	All uppercase; words separated by underscore	<code>PI</code> , <code>MAX_VALUE</code>

Table 3.10: Java naming conventions

- Names starting with `java` and `javax` are reserved for the standard libraries.
- A package name must follow the rules of identifiers: no spaces, cannot be a keyword, cannot start with a digit.
- Avoid using uppercase in package names to prevent clashes with class names.

3.2.5 Creating a Package

To create a user-defined package, the `package` statement is written as the **very first statement** of the source file (comments excepted), followed by the class definition. The general form is:

Syntax

```
package packageName;
public class ClassName
{
    // body
}
```

Steps for creating a package

- 1 Declare the package at the top of the source file with the `package` keyword.
- 2 Define the class (or interface) that is to be placed in the package and declare it **public** so that it can be used from outside.
- 3 Create a sub-directory with the **same name as the package** (e.g. folder `package1`), and save the file in it as `ClassName.java`.
- 4 Compile the file so that the `.class` file is stored in the package folder. The simplest way is: `javac -d . ClassA.java`
- 5 The `-d` option tells `javac` where to place the output; the dot (`.`) means the current directory, and `javac` automatically creates the package folder if it does not exist.

Example 7: ClassA.java in package package1

```
package package1;
public class ClassA
{
    public void displayA()
    {
        System.out.println("Class A");
    }
}
```

Example 8: ClassB.java in package package2

```
package package2;
public class ClassB
{
    protected int m = 10;
    public void displayB()
    {
        System.out.println("Class B");
        System.out.println("m = " + m);
    }
}
```

Directory structure

Figure 3.1: Folder layout of the packages

```
Project folder
|
|-- PackageTest1.java          (uses the packages)
|-- PackageTest1.class
|
|-- package1
|   |-- ClassA.java
|   |-- ClassA.class
|
|-- package2
|   |-- ClassB.java
|   |-- ClassB.class
```

- There can be **only one package statement** in a source file and it must come first.
- If the package statement is omitted, the classes belong to the **default (unnamed) package** of the current directory.
- A package such as **a . b . c** needs the folder structure **a/b/c**.
- A source file can contain **only one public class**, and the file name must match its name.

3.2.6 Accessing a Package

A package can be accessed from another program in the same three ways described for system packages: by **import of a single class**, by **import of all classes** with the asterisk, or by the **fully qualified name**. The general form of the import statement is:

Syntax

```
import package1[.package2][.package3].classname;  
import package1.*;
```

- **package1** is the top-level package; **package2** etc. are sub-packages inside it; **classname** is the class to be imported.
- The import statements must be written **after the package statement and before any class definition**.
- For the compiler and JVM to find a package, the folder containing the package folder must be in the **current directory** or in the **CLASSPATH**.
- The classpath can be set with the option **-cp** (or **-classpath**), e.g. `java -cp C:\myclasses PackageTest1`, or through the **CLASSPATH** environment variable.
- A class is run with its fully qualified name when it is inside a package, e.g. `java package1.Main`, from the folder that contains package1.

3.2.7 Using a Package

Once the packages are created and compiled, any program can use their public classes. The program below imports ClassA from package1 and ClassB from package2 and calls their public methods.

Example 9: PackageTest1.java

```
import package1.ClassA;  
import package2.ClassB;  
class PackageTest1  
{  
    public static void main(String[] args)  
    {  
        ClassA objectA = new ClassA();  
        ClassB objectB = new ClassB();  
        objectA.displayA();  
        objectB.displayB();  
    }  
}
```

Output

```
Class A  
Class B  
m = 10
```

Only **public** classes and members can be reached from another package. In the example above, the protected variable `m` cannot be accessed directly by PackageTest1 (writing `objectB.m` would be an error) but it is used inside ClassB's public method.

Access protection in packages

Java provides four levels of access for class members. The table shows where each modifier allows access.

Visibility	private	default (no modifier)	protected	public
Same class	Yes	Yes	Yes	Yes
Same package: subclass	No	Yes	Yes	Yes
Same package: non-subclass	No	Yes	Yes	Yes
Different package: subclass	No	No	Yes	Yes
Different package: non-subclass	No	No	No	Yes

Table 3.11: Access protection of class members

Note: A top-level class can be either public or default (package-private). Private and protected apply only to members and nested classes.

Handling name clashes

If two packages contain classes of the same name and both are imported with the asterisk, the compiler cannot decide which one is meant. The fully qualified name removes the doubt:

Example 10: Same class name in two packages

```
import java.util.*;
import java.sql.*;
public class Clash
{
    public static void main(String[] args)
    {
        java.util.Date d1 = new java.util.Date();
        java.sql.Date d2 = new java.sql.Date(0L);
    }
}
```

3.2.8 Adding a Class to a Package

It is simple to add a new class to an existing package. The new class just has to carry the same package statement as the other classes of that package.

Steps

- 1 Open or decide the package in which the class is to be placed (for example package1).
- 2 Write the **same package statement** at the top of the new source file.
- 3 Define the new class and declare it **public** if it is to be used outside the package. The file name must be the class name.
- 4 Place the file in the package folder (or compile with -d) and compile it.
- 5 The resulting .class file is stored along with the other classes of the package.

Example 11: Adding ClassC to package1

```
package package1;
public class ClassC
{
    public void displayC()
    {
        System.out.println("Class C");
    }
}
```

Compilation command

```
javac -d . ClassC.java
```

After compilation, package1 contains ClassA.class and ClassC.class, and both can be imported using `import package1.*;`

- A package can have **many public classes**, but each must be in its own file.
- If a file contains several classes, only one may be public; the rest are package-private.
- When a package is shared by many files, all files must start with the identical package statement.
- Programs that use the package can later be compiled and executed without any change to existing classes.

Comparison: using a package vs not using a package

Without packages	With packages
All classes sit in one folder; difficult to manage	Classes are grouped logically in folders
Class name clashes are likely	Same class name can exist in different packages
No access control beyond the class	Package-level protection using protected and default
Hard to reuse in other projects	Easy to import and reuse

Table 3.12: Benefits of packages

Points to remember: Packages

- The package statement must be the first statement in the file; there can be only one.
- `java.lang` is automatically imported; all other packages need import.
- `import java.util.*` does not import sub-packages.
- A package name equals a folder path on disk (dots become folder separators).
- Compile with `javac -d . File.java` to create the folders automatically.
- Only public classes and members are available outside the package.
- Package names are lowercase and use the reversed domain name for uniqueness.

Multiple Choice Questions (15)

Choose the correct option. The answer key is given at the end of this section.

Q1. Which keyword is used to define an interface in Java?

- (A) class
- (B) interface
- (C) implements
- (D) abstract

Q2. Variables declared inside an interface are implicitly:

- (A) private static
- (B) protected final
- (C) public static final
- (D) public abstract

Q3. Before Java 8, methods declared in an interface are implicitly:

- (A) public abstract
- (B) private
- (C) protected static
- (D) final

Q4. Which statement about interfaces is TRUE?

- (A) Objects of an interface can be created with new
- (B) An interface can have constructors
- (C) An interface cannot be instantiated
- (D) An interface can have instance variables

Q5. A class uses which keyword to accept an interface?

- (A) extends
- (B) implements
- (C) inherits
- (D) interface

Q6. Which of the following is the correct way for an interface to inherit from two interfaces?

- (A) interface C implements A, B
- (B) interface C extends A, B
- (C) interface C inherits A, B
- (D) interface C extends A implements B

Q7. If a class does not implement all abstract methods of an interface, the class must be declared:

- (A) final
- (B) static
- (C) abstract
- (D) private

Q8. Multiple inheritance in Java is achieved through:

- (A) Multiple classes
- (B) Interfaces
- (C) Packages
- (D) Constructors

Q9. Which is the correct syntax for a class that extends a class and implements two interfaces?

- (A) class C implements I1, I2 extends B
- (B) class C extends B implements I1, I2
- (C) class C extends B, I1, I2
- (D) class C implements B extends I1, I2

Q10. The package statement, if present, must be:

- (A) The last statement in the file
- (B) After the class definition
- (C) The first statement in the file (excluding comments)
- (D) Inside the main method

Q11. Which package is imported automatically into every Java program?

- (A) java.util
- (B) java.io
- (C) java.lang
- (D) java.awt

Q12. Which command compiles a file and creates the package folder automatically?

- (A) java -d . File.java
- (B) javac -d . File.java
- (C) javac -p . File.java
- (D) javap File.java

Q13. The statement `import java.util.*;` imports:

- (A) Only the Scanner class
- (B) All classes of java.util including its sub-packages
- (C) All classes directly inside java.util but not its sub-packages
- (D) Nothing; it is an error

Q14. A class member with NO access modifier (default) is accessible:

- (A) Only in the same class
- (B) Within the same package
- (C) Everywhere
- (D) Only in subclasses of other packages

Q15. According to the naming convention, package names are written in:

- (A) UPPERCASE
- (B) lowercase
- (C) Capital first letters of every word
- (D) snake_case with capitals

Answer Key

Q	Ans	Q	Ans	Q	Ans
1	B	6	B	11	C
2	C	7	C	12	B
3	A	8	B	13	C
4	C	9	B	14	B
5	B	10	C	15	B

Table: Answer key for the 15 MCQs

Short explanations

- **Q1 (B):** The keyword interface starts an interface definition.
- **Q2 (C):** Interface variables are always public static final (constants).
- **Q3 (A):** Interface methods are public and abstract unless they are default, static or private.
- **Q4 (C):** Interfaces are incomplete by nature; they cannot be instantiated.
- **Q5 (B):** A class implements an interface; an interface extends another interface.
- **Q6 (B):** Interfaces use extends, with several parents separated by commas.
- **Q7 (C):** An abstract class is allowed to leave abstract methods undefined.
- **Q8 (B):** Java does not allow multiple class inheritance, so interfaces are used.
- **Q9 (B):** The extends clause comes first, then implements.
- **Q10 (C):** package must precede import statements and class definitions.
- **Q11 (C):** java.lang is imported by the compiler automatically.
- **Q12 (B):** The -d option places class files in package folders and creates them.
- **Q13 (C):** The asterisk imports classes of that package only, not sub-packages.
- **Q14 (B):** Default access means package-private visibility.
- **Q15 (B):** Package names are conventionally all lowercase.

Broad Questions (10)

Answer the following questions in detail. Where applicable, write syntax, program and output.

- Q1.** What is an interface? Explain how it differs from a class. Describe how to define an interface with syntax and a suitable example.
- Q2.** How does Java achieve multiple inheritance? Explain with a program in which a class extends one class and implements an interface.
- Q3.** Explain 'extending interfaces' in detail. Write a program in which an interface extends two other interfaces and is implemented by a class.
- Q4.** Describe how a class implements one or more interfaces. State the rules to be followed. Write a program that uses an interface Area for the classes Rectangle and Circle.
- Q5.** Explain how interface variables are accessed. Why are they called constants? How is the ambiguity of same-named variables in two interfaces resolved? Give an example.
- Q6.** Define a package. List the advantages of packages. Explain the Java system packages in a table with their purpose and important classes.
- Q7.** Explain the different ways of using system packages in a program using import statements, with examples. Why is java.lang special?
- Q8.** Explain the naming conventions followed in Java for packages, classes, interfaces, methods, variables and constants. Why is the reversed domain name used for packages?
- Q9.** Explain the complete procedure to create a user-defined package, compile it, access it from another program and run the program. Show the directory structure and use of -d and CLASSPATH.
- Q10.** Explain access protection in packages using a table of private, default, protected and public. Also explain the steps to add a new class to an existing package with an example.

End of Chapter 3: Interfaces and Packages