1. What are packages in Java? Explain with an example

Packages in Java are used to group related classes together and help keep the code organized and manageable. They also help avoid name conflicts between classes.

Think of a package like a folder in your computer that stores related files together.

☑ Why Use Packages?
To organize your classes neatly.

To prevent name conflicts (e.g., two classes named List can exist in different packages).

To reuse code more easily.

☑ Syntax to Create a Package:

```
package mypackage;
```

☑ Example:

🗀 Folder Structure:

```
MyProject/
├── mypackage/
│   └── MyClass.java
└── MainClass.java
```

mypackage/MyClass.java

```
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass in mypackage");
    }
}
```

MainClass.java

```
import mypackage.MyClass;
```

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

Write a short note on access protection in java packages

Access protection in Java controls who can access classes, methods, and variables. It helps in encapsulation and keeps code secure and organized.

Java provides four types of access levels:

public – Can be accessed from anywhere.

private – Can be accessed only within the same class.

protected – Can be accessed in the same package and by subclasses in other packages.

default (no keyword) – Can be accessed only within the same package

☑ Based on where the class is:
Same package & subclass → can access protected/default/public.

Same package & non-subclass → can access default/public.

Different package & subclass → can access protected/public.

Different package & non-subclass → can access only public.

3. Explain the types of Java packages?

1. Built-in Packages
These are predefined packages provided by Java.

They contain commonly used classes and interfaces.

◇ Examples:
| **Package** | **Why It’s Important**

`java.lang`	Core language support. Always imported automatically. Contains `String`, `Math`, `System`, etc. — very frequently asked.
`java.util`	Covers **Collections**, `ArrayList`, `HashMap`, `Date`, etc. Often used in coding and theory questions.
`java.io`	File handling and input/output. Frequently appears in practical and theory questions.
`java.sql`	Database connectivity using JDBC. Important for questions related to DB operations.
`javax.swing` or `java.awt`	Used for GUI programming. Basic knowledge is expected, especially in app development-related questions.

2️⃣ User-defined Packages
These are packages created by the programmer to organize their own classes.

```
package mypackage;

public class MyClass {
    public void show() {
        System.out.println("Hello from MyClass");
    }
}

import mypackage.MyClass;
```

4. Write the Differences between Interface and Class in Java

Aspect	**Interface**	**Class**
1. Definition	A blueprint that contains **abstract methods** and constants.	A template that defines **properties and behavior** (data + methods).

2. Keyword Used	Defined using the `interface` keyword.	Defined using the `class` keyword.
3. Method Implementation	Methods are **abstract** (by default, until Java 7).	Methods can be **fully defined and implemented**.
4. Variables	Variables are **public, static, and final** by default.	Variables can be of **any access type** and not necessarily final.
5. Inheritance	A class can **implement** multiple interfaces.	A class can **extend only one class** (single inheritance).
6. Access Modifiers	Methods are **public** by default.	Methods can have **private, protected, or public** modifiers.
7. Object Creation	Cannot create an object of an interface.	Can create objects of a class (unless it's abstract).
8. Purpose	Used to define **what should be done** (contract).	Used to define **how things should be done** (implementation).
9. Constructors	Interfaces **do not have constructors**.	Classes **can have constructors**.
10. Use Case	Useful for **achieving abstraction and multiple inheritance**.	Useful for **code reuse and defining actual behaviors**.

Use interfaces to define rules or capabilities (what to do).

Use classes to define complete behaviors (how to do it).

5. Write a Java program to demonstrate accessing of members when corresponding classes are imported and not imported.

☑ Goal: Show what happens when:
A class is imported from a package.

A class is not imported, and we access it using the fully qualified name.

🗀 Folder Structure:

```
AccessDemo/
├── mypackage/
│   └── MyClass.java
└── Demo.java
```

mypackage/MyClass.java

```
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Accessed MyClass from mypackage!");
    }
}
```

Demo.java

```
// Uncomment the line below to test importing
// import mypackage.MyClass;

public class Demo {
    public static void main(String[] args) {

        // Case 1: Accessing without import using fully qualified name
        mypackage.MyClass obj1 = new mypackage.MyClass();
        obj1.display();

        // Case 2: Accessing with import (uncomment import and lines below to
test)
        /*
        MyClass obj2 = new MyClass();
        obj2.display();
        */
    }
}
```

☑ Explanation:
In Case 1, we access the class using the full package name: mypackage.MyClass – works without import.

In Case 2, we import the class and use it directly as MyClass. You must uncomment the import line and related code to use this.

Output:

```
Accessed MyClass from mypackage!
```

13. Explain ‘throw’, ‘throws’ and ‘finally’ keywords for handling exceptions in Java

◇ 1. throw

Used to manually throw an exception.

Can throw checked or unchecked exceptions.

◇ Syntax:

```
throw new ArithmeticException("Divide by zero error");
```

◇ Example:

```
public class Example {
    public static void main(String[] args) {
        throw new NullPointerException("Manually thrown");
    }
}
```

2. throws

Used in method definition to declare exceptions that the method might throw.

It tells the caller of the method to handle the exception.

◇ Syntax:

```
void myMethod() throws IOException {
    // code that may throw IOException
}
```

◇ Example:

```
import java.io.*;

public class Example {
    void readFile() throws IOException {
        FileReader fr = new FileReader("file.txt");
    }
}
```

◇ 3. finally

Used to define a block of code that always executes, whether an exception occurs or not.

Usually used to release resources (like closing files or connections).

◇ Syntax:

```
java

try {
    // risky code
```

```
} catch (Exception e) {
    // handle exception
} finally {
    // always executes
}
```

◇ Example:

```
public class Example {
    public static void main(String[] args) {
        try {
            int a = 5 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

Keyword	Purpose
`throw`	Manually throws an exception
`throws`	Declares exception in method signature
`finally`	Always executes, used for cleanup

11. Define the following: (i) Try (ii) catch (iii) throw (iv) super

i) try
Used to wrap code that might cause an exception.

It must be followed by either catch or finally.

java
Copy code
```
try {
    int a = 5 / 0;
}
```
(ii) catch
Used to handle exceptions thrown in the try block.

java

Copy code
```
catch (ArithmeticException e) {
    System.out.println("Error: " + e);
}
```
(iii) throw
Used to manually throw an exception.

java
Copy code
```
throw new NullPointerException("Manual throw");
```

12. Interfacing

```
interface Animal {
    void eat();
}

interface Pet extends Animal {
    void play();
}

class Dog implements Pet {
    public void eat() {
        System.out.println("Dog is eating.");
    }

    public void play() {
        System.out.println("Dog is playing.");
    }
}

public class ExtendInterfaceDemo {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.play();
    }
}
```

Write a Java program to show how to import packages and interfaces in Java

Write a program to import classes from user defined packages

☑ Folder Structure:
Copy code

```
MyProject/
├── mypackage/
│   └── Pet.java
│   └── Dog.java
└── MainDemo.java
```

◇ mypackage/Pet.java

```
package mypackage;

// Interface
public interface Pet {
    void play();
}
```

◇ mypackage/Dog.java

```
package mypackage;

// Class implementing the interface
public class Dog implements Pet {
    public void play() {
        System.out.println("Dog is playing.");
    }
}
```

◇ MainDemo.java

```
import mypackage.Dog;     // Import class
import mypackage.Pet;     // Import interface

public class MainDemo {
    public static void main(String[] args) {
        Dog myDog = new Dog();  // Using imported class
        myDog.play();           // Calling method from interface
    }
}
```