Module-3:

INTERFACES AND PACKAGES

PRAMOD NAIK

What is an interface?

- In Java, an **interface** is a **reference type**, similar to a **class**, that can contain only
  1. **public and abstract methods** (Starting with Java 8, you can also have **default** and **static** methods in an interface)
  2. **public static final fields**.
  3. It **cannot have Constructor**
- Interfaces specify **what a class must do**, **but not how it does it**.
- A class that **implements** an interface must **implement all of its methods unless it is an abstract class**.
- Interfaces are used to achieve:
  - **Abstraction**,
  - **Multiple Inheritance**, and
  - **Loose coupling** between components in Java.

Data Abstraction

Abstract Classes and Interfaces

PRAMOD NAIK

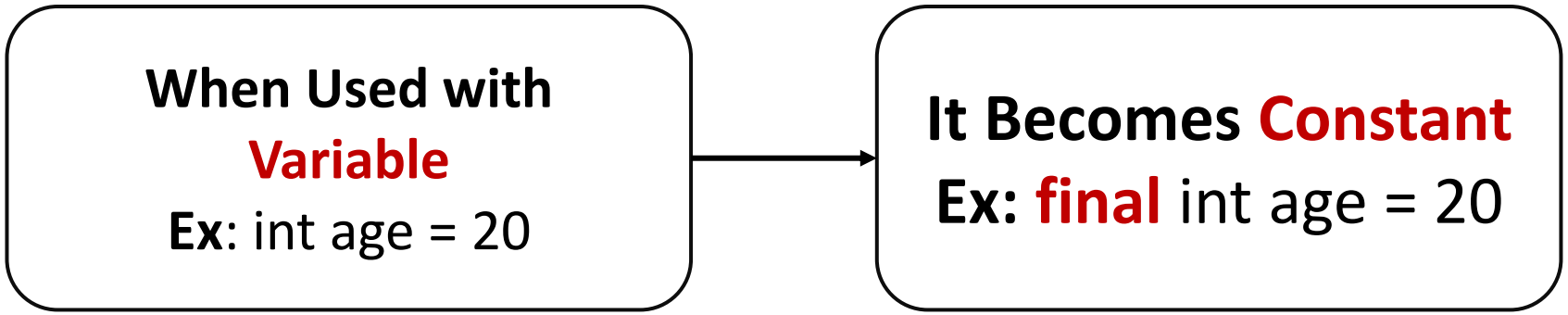
What is Abstract Method:

An abstract method is a method declared **without a body**. It must be overridden in the subclass. The method signature includes the **abstract** keyword, and there is no method body.

Key Rules for Abstract Methods:

1. Must be declared in an **abstract class.**
2. Cannot have a **method body.**
3. Subclasses that extend the abstract class must **override** the abstract method.

final Fields (Constants): When we use a final Keywork with a variable it becomes Constant or final Variable.



What is Abstract Class:

An **Abstract Class** in object-oriented programming is a class that **cannot be instantiated directly**. It serves as a **blueprint for other classes**. Abstract classes are used when you want to define some common behavior (methods) that other classes should inherit and implement, but the abstract class itself should not be instantiated. In other words, an abstract class can have **abstract methods** (without implementation) and/or **concrete methods** (with implementation).

Defining and Creating an interface:

1. **Declaration:** An interface is declared using the **interface keyword**:

```
public interface Animal {
    int AGE = 0;  // Constant variable

    void eat();   // Abstract method
    void sleep(); // Abstract method
}
```

Note:

By **default** all the Methods inside an interface is **Abstract methods** and also all the fields in inside an interface are **final** or **constant**.

2. Implementation by a Class:

A class can implement an interface by using the **implements keyword**. The class must **provide implementations** for all of the interface's abstract methods.

Example:

In this example, Dog implements the Animal interface and provides concrete implementations of the eat() and sleep() methods.

```java
public class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog eats bones");
    }

    @Override
    public void sleep() {
        System.out.println("Dog sleeps in the kennel");
    }
}
```

PRAMOD NAIK

Accessing implementations through interface references:

This means that you **interact with an object of a class that implements an interface** via **a reference of that interface type**, not the class type. This allows you to use the object polymorphically, meaning that the actual implementation is determined at runtime, but you interact with it using the common interface methods.

Example:

1. Below Code is for Creating an **interface** with name **Shape**

```
// Interface definition
public interface Shape {
    void draw();
}
```

2. **Class** Circle **implements** the **Interface** Shape:

```
// Implementations of the interface
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

3. **Class** Rectangle **implements** the **Interface** Shape:

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

Main Class:

```
// Main class to demonstrate interface reference
public class Main {
    public static void main(String[] args) {
        // Accessing Circle through Shape interface reference
        Shape shape1 = new Circle();
        shape1.draw(); // Output: Drawing a circle

        // Accessing Rectangle through Shape interface reference
        Shape shape2 = new Rectangle();
        shape2.draw(); // Output: Drawing a rectangle

        // Accessing Triangle through Shape interface reference
        Shape shape3 = new Triangle();
        shape3.draw(); // Output: Drawing a triangle
    }
}
```

Extending interface:

In Java, **interfaces** can **extend** other **interfaces**, similar to how **classes** can **extend** other **classes**.

When one interface extends another, it **inherits** all the abstract methods, default methods, and constants (static and final fields) from the parent interface. This allows you to create a more specialized interface by building upon an existing one.

Unlike classes, where a subclass can only extend **one parent class** (due to **single inheritance**), an interface in Java can **extend** **multiple interfaces**, allowing for a form of **multiple inheritance**.

Syntax:

```
interface InterfaceA {
    void methodA();
}

interface InterfaceB {
    void methodB();
}

interface ChildInterface extends InterfaceA, InterfaceB {
    void methodC();
}
```

Interfaces VS **Abstract classes:**

In Java, both **interfaces** and **abstract classes** are used to define abstract types, which **cannot be instantiated** directly.

Data Abstraction -> **Abstract class** & **Interfaces**.

Feature	Interface	Abstract Class
Created Using	**interface** Keyword	**abstract** keyword
Multiple Inheritance	Yes	No
Default Methods	Yes (Java 8+)	No (but can have concrete methods)
Static Methods	Yes (Java 8+)	Yes
Fields	Public, static, and final only	Any type of field
Constructors	No	Yes (cannot instantiate directly)
Method Modifiers	Public (Java 9+ allows private methods)	Can be public, private, protected
Abstract/Concrete Methods	Abstract, default, or static	Both abstract and concrete methods
Use Case	To define capabilities (behavior)	To define a base class (object template)
Inheritance	A class can implement multiple interfaces	A class can extend only one class

Encapsulation

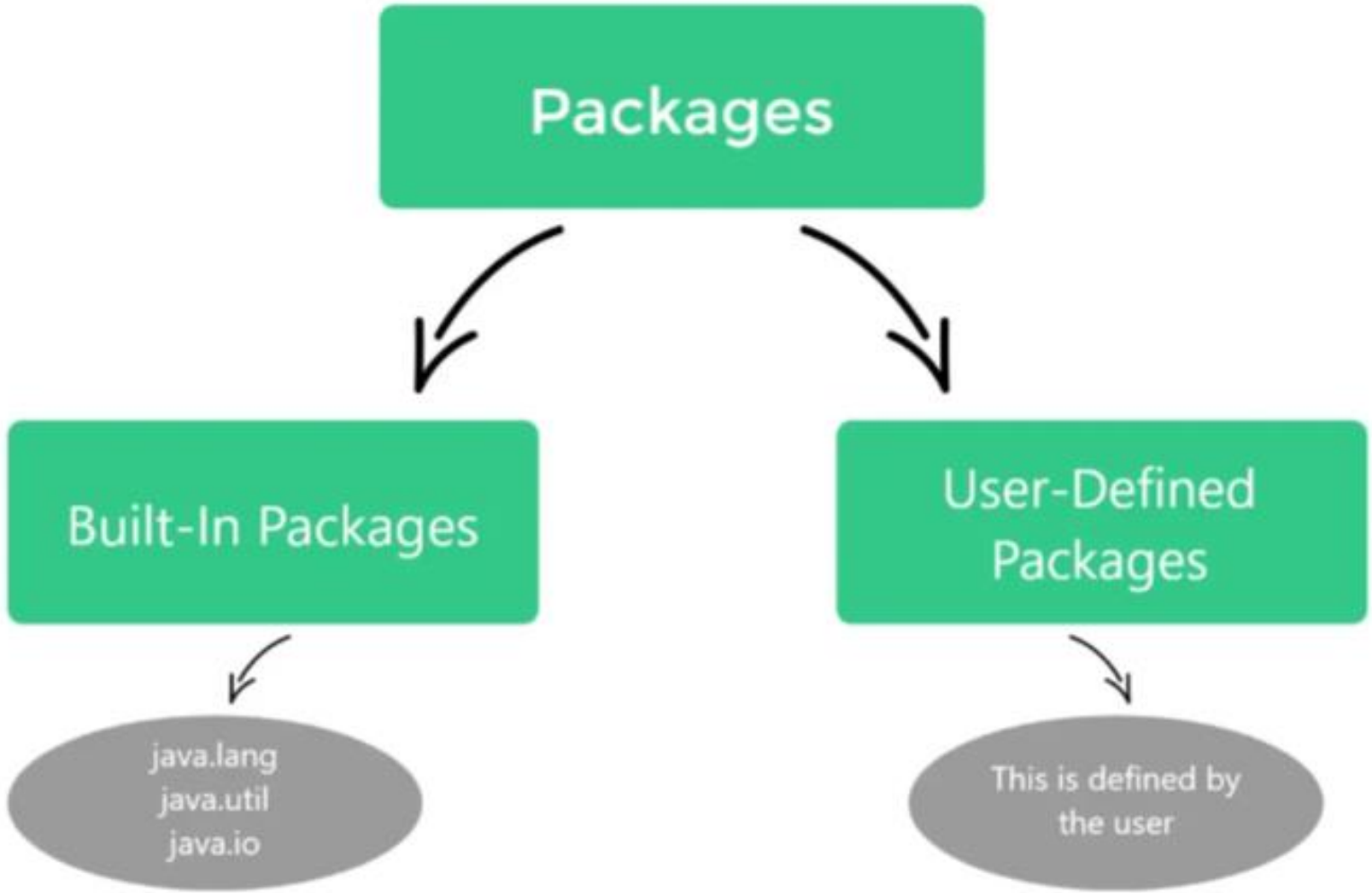
Encapsulation is a key Java OOP principle that is implemented mostly through by making a class Property **Private** (Getters and Setters) and **packages**.

packages :

- The packages in Java Language is nothing more than **directories** that gathers **related classes**, **interfaces**, and **sub-packages** based on their **functional relationships** in java.
- A large project's classes can be organised using **Java packages**, which can aid in the **implementation of encapsulation.**

Packages in Java

2-Types



PRAMOD NAIK

1. pre-defined packages:

In **java**, we already have a number of pre-defined packages that contain a large number of **classes** and **interfaces**. These packages are known as Built-in packages and we can directly import them in our code.

Some commonly used built-in packages are:

1. **java.io:** Data streams, serialisation, and the file system are all supported by this package for system input and output.
2. **java.lang:** contains classes that are essential to the design of the Java programming language.
3. **java.math:** Classes in this package perform arbitrary-precision integer arithmetic (BigInteger) and arbitrary-precision decimal arithmetic (BigDecimal).
4. **java.util:** The collections framework, some internationalisation support classes, and a few utility classes are included.

How to use Built-In Packages:

```
import java.util.*;
or
import java.util.Scanner;
```

2. User-Defined Packages:

Java also allows you to make your own packages. These are referred to as user-defined packages.

1. Creating a Package: To Create a package use the package keyword at the top of your file, the automatically all the class below the package name will be placed inside of that package.

```
package demoPackage;
    public class demoClass
    {
        public void getMessage(String m)
        {
            System.out.println(m);
        }
    }
```

2. Accessing a Package:

We can access all the classes and interfaces inside of any package by using **import Keyword** or **Fully Qualified Class Name**.

1. Importing a Package Using import Keyword:

- **Importing a Specific Class**: If you want to import a specific class from a package, use:

```
import packageName.ClassName;
```

- **Importing All Classes from a Package:** If you want to import all classes from a package, use a wildcard *:

```
import packageName.*;
```

2. Using Fully Qualified Class Name:

```
java.util.Scanner scanner = new java.util.Scanner(System.in);
```

Points to Remember:

- Either **import the class** or use the **fully qualified class name** in order to access code that is located ***outside the current package***.
- Using packages while coding provides code **reusability benefit**.
- Packages allow us to **uniquely identify a class.**
- Access control features include protected classes, default classes, and private classes.
- They enable the **hiding of classes**, preventing other programmes from accessing classes intended for internal use only.
- You can better **organise your project** and find related classes by using packages.

Understanding CLASSPATH:

The **CLASSPATH** in Java is a critical concept that helps the **Java Virtual Machine (JVM)** and the **compiler** find and load **compiled** class files (.class) and **external libraries** (typically **.jar files**) that your application depends on.

The CLASSPATH is an **environment variable** (or can be specified as a command-line option) that specifies the **locations where the JVM and the Java compiler should look for .class files and libraries.**

These locations can be:

- **Directories containing .class files**.
- **JAR** (Java ARchive) files containing libraries.
- **ZIP** files containing compiled Java code.

By default, the JVM looks in the **current directory** for class files, but the CLASSPATH allows you to tell Java to look in other directories or files as well.

Why is CLASSPATH Important?

When you compile or run a Java program, the **JVM needs to know where to find the classes that your code references**. If the JVM can't locate a class, it will throw a **ClassNotFoundException** or a **NoClassDefFoundError**.

For example, if your project relies on **external libraries** or **classes** that are stored in different locations, you need to include these paths in the **CLASSPATH** so that Java knows where to find them.

You can set the CLASSPATH in two ways:

- **Temporarily:** via the command line when compiling and running a Java program.
- **Permanently:** by setting it as an **environment variable.**

Setting CLASSPATH in Command Line

You can specify the classpath using the **-cp** or **-classpath** option when running the **javac** (compiler) or **java** (JVM) commands.

For example, to compile and run a Java program that depends on classes in a directory or **JAR file**:

Example:

```
java -cp C:\path\to\classes;C:\path\to\library.jar MyProgram
```

PRAMOD NAIK