**Module-3: INTERFACES AND PACKAGES**

**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. I**t** **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:
  1. **Abstraction**,
  2. **Multiple Inheritance**, and
  3. **Loose coupling** between components in Java.

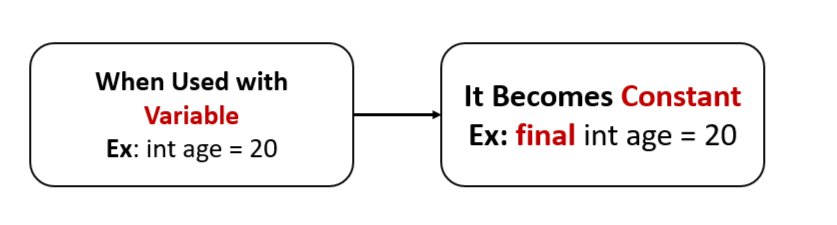
**What is Abstract Method:**

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**:

**A screen shot of a computer code

Description automatically generated**

**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**.

1. **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:**

A computer code on a black background

Description automatically generated

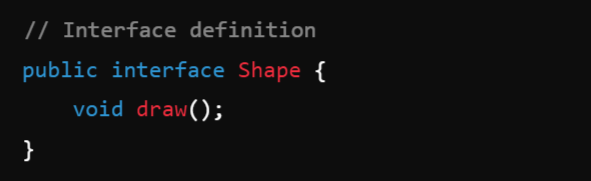
In this example, Dog implements the Animal interface and provides concrete implementations of the eat() and sleep() methods.

**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**



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

A computer code on a black background

Description automatically generated

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

A computer code on a black background

Description automatically generated

**Main Class:**

A computer screen shot of a black screen with white text

Description automatically generated

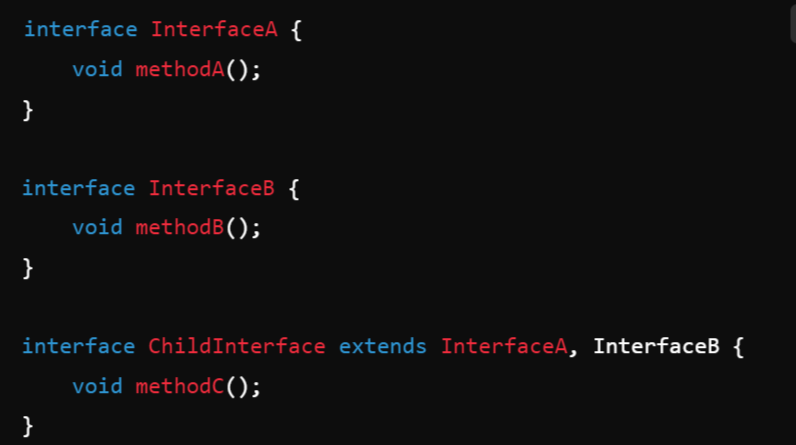
**Extending interface:**

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**.

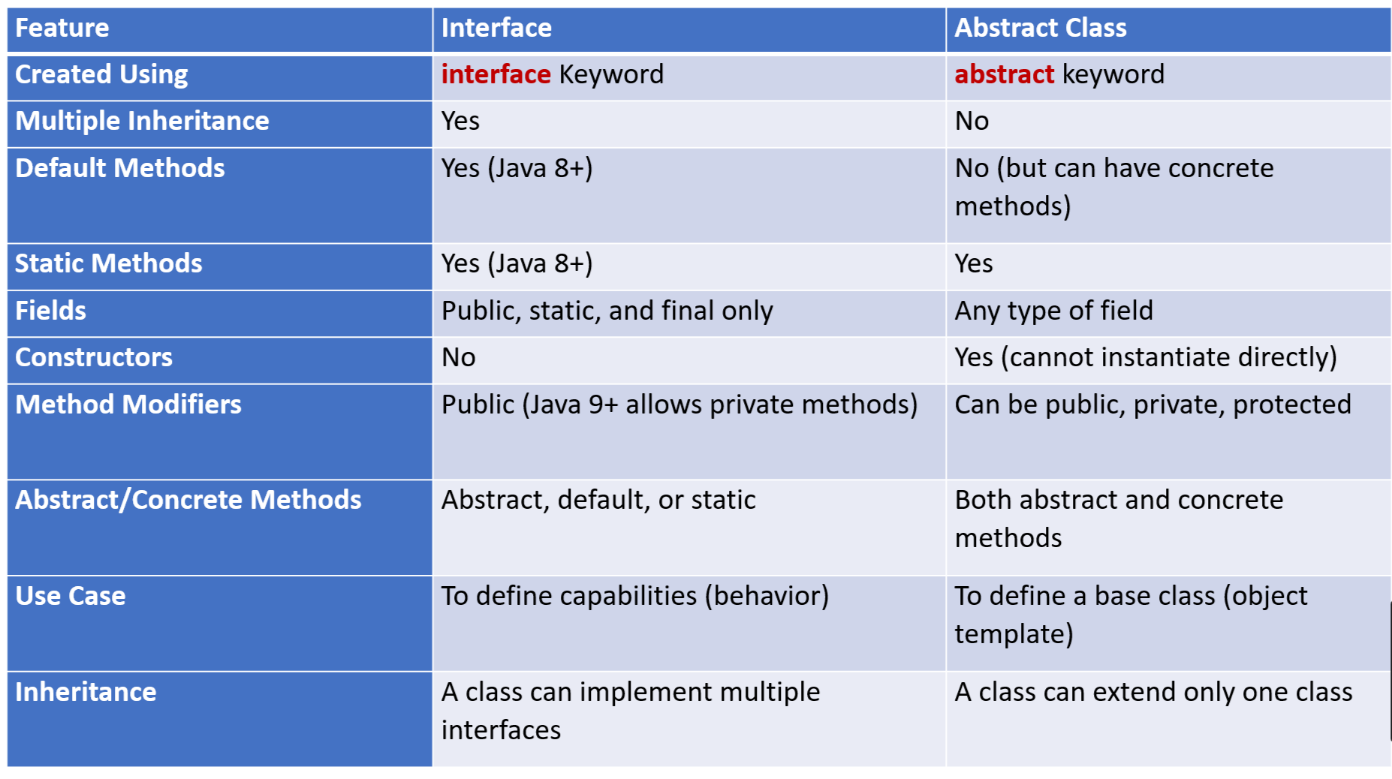
**Syntax:**

****

**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.**



**What is Multiple Inheritance?**

Multiple inheritance is a feature in object-oriented programming (OOP) where a **class can inherit features (properties and methods) from more than one parent class**. This means that a derived class can have more than one base class, inheriting functionalities from multiple sources.

In multiple inheritance:

* The child class gains properties and behaviors (methods) from all its parent classes.
* It provides more flexibility, but it also **introduces complexity**, especially when the same method or property is defined in multiple parent classes. This ambiguity is known as the **Diamond Problem.**

**Why is Multiple Inheritance not directly supported in Java?**

Java does not allow multiple inheritance with classes to **avoid** the **Diamond Problem** and simplify the design of the language. The Diamond Problem occurs when two classes (B and C) inherit from a common superclass (A), and a class D inherits from both B and C. If class A has a method methodA(), and both B and C override this method, it's unclear which version of methodA() should be used in class D.

To **prevent this complexity**, Java allows multiple inheritance only through **interfaces**, not through classes. An interface defines methods but does not provide their implementation (before Java 8). Multiple interfaces can be implemented without **ambiguity** because each implementing class must provide its own method implementation, thus resolving any conflicts.

**Example of Multiple Inheritance using Interfaces in Java**

Here’s an example to demonstrate how Java handles multiple inheritance using interfaces:  
  
**Write a JAVA program to demonstrate Inheritance. Sample Program on Java for the implementation of Multiple inheritance using interfaces to calculate the area of a rectangle and triangle.**



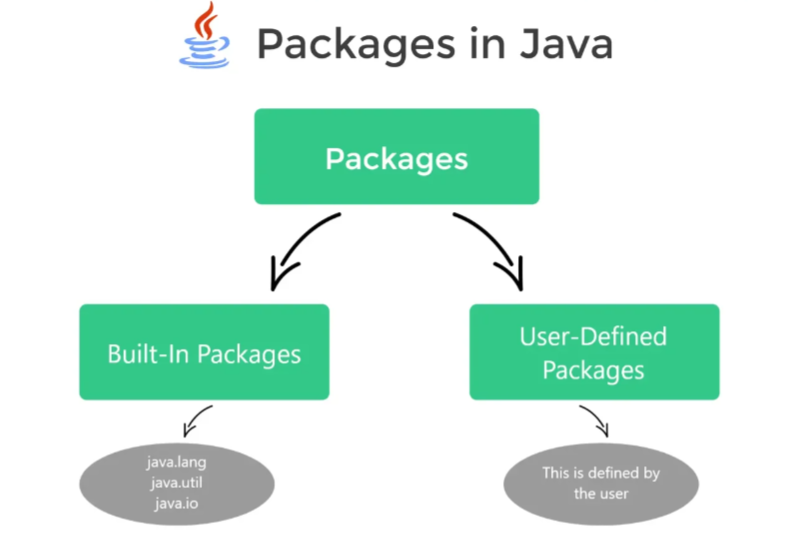
**Encapsulation**

**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.**

**2-Types:**



**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:**

A screen shot of a computer screen

Description automatically generated

**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.

A computer screen shot of a program code

Description automatically generated

**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:

A black background with white text

Description automatically generated

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

A black background with white text

Description automatically generated

**2. Using Fully Qualified Class Name:**



**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 package es 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:**

