A key reason to use Java?

Is to tackle concurrent programming. With Parallel Algorithms and threadsafe data structures readily avaialable in the java library.

Package? A Package is a namespace that organizes a set of related classes, interfaces, and sub-packeges.

- It helps to avoid `name conflicts`.
- Makes code more `Modular`.
- Easy to maintain.
- Logically grouped.

Ex: com.example.math.Calculator Ex: com.example.finance.Calculator

Stack Diagram? - A Stack diagram is memory diagram that shows currently running methods. - It helps to visualize the `scope of the variable`. - It's a good mental model for how `variables and methods` work at `runtime`. - We can even trace the execution of the program on a paper or a whiteboard(smaller programs).

Wrapper Class

For every primitive type in Java, there is a built-in object type called a wrapper class. The wrapper class for int is called Integer, and for double it is called Double.

Why?

Sometimes you may need to create a wrapped object for a primitive type so that you can give it to a method that is expecting an object.

Autoboxing

Autoboxing is the automatic conversion that the Java compiler makes between primitive types and their corresponding object wrapper classes. This includes converting an int to an Integer and a double to a Double.

The Java compiler applies autoboxing when a primitive value is:

Passed as a parameter to a method that expects an object of the corresponding wrapper class.

Assigned to a variable of the corresponding wrapper class. Ex:

```
Integer i = 10; // Javac applies Autoboxing.
```

Unboxing

Unboxing is the automatic conversion that the Java compiler makes from the wrapper class to the primitive type. This includes converting an Integer to an int and a Double to a double.

The Java compiler applies unboxing when a wrapper class object is:

Passed as a parameter to a method that expects a value of the corresponding primitive type.

Assigned to a variable of the corresponding primitive type. Ex:

```
int val = i; // Javac applies Unboxing.
```

OOP

Inheritance:

Inheritance is a process where one class acquires the properties (methods and attributes) of another class.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

Super Class –> Chil Class

Why?

This allows for code reusability and hierarchical organization of classes.

Types of Inheritance:

1. Single
2. Multilevel
3. Hierarchical
4. Multiple
5. Hybrid

Single-Inherirance

Single inheritance means a class (called the child or subclass) inherits properties and behaviors (methods and fields) from only one parent class (also called the superclass).

```
class Parent {
        void show() { System.out.println("Parent method"); }
}
class Child extends Parent {
```

```java
        void display() { System.out.println("Child method"); }
}
```

Child inherits from Parent using the extends keyword.

Multilevel-Inheritance:

Multilevel inheritance occurs when a class extends another class, which in turn extends another class, forming a chain. This allows a class to inherit from a chain of superclasses.

Child –> Parent –> Grandparent

```java
class Grandparent {
    void grandparentMethod() { System.out.println("Grandparent"); }
}
class Parent extends Grandparent {
    void parentMethod() { System.out.println("Parent"); }
}
class Child extends Parent {
    void childMethod() { System.out.println("Child"); }
}
```

Here, `Child` inherits from `Parent`, and `Parent` inherits from `Grandparent`. So, `Child` can access methods from both `Parent` and `Grandparent`.

Hierarchical Inheritance:

Hierarchical inheritance happens when multiple classes inherit from the same parent class. This means there is one superclass and multiple subclasses.

```java
class Animal {
    void eat() { System.out.println("Eating..."); }
}
class Dog extends Animal {
    void bark() { System.out.println("Barking..."); }
}
class Cat extends Animal {
    void meow() { System.out.println("Meowing..."); }
}
```

In this example, both Dog and Cat inherit from Animal. This setup allows each child class to reuse code from the parent class.

Inheritance Type	Description	Example Structure
Single	One child class inherits from one parent class	`B extends A`
Multilevel	Chain of inheritance: child → parent → grandparent	`C extends B extends A`
Hierarchical	One parent class, multiple child classes	`B extends A`, `C extends A`

Multiple-Inheritance:

Java does not support multiple inheritance with classes—meaning a class cannot extend more than one class at a time.

However, Java does allow a form of multiple inheritance using interfaces. A class can implement multiple interfaces, and because interfaces only define method signatures (not implementations), there is no ambiguity when the implementing class provides its own method implementations.

`Example: Multiple Inheritance Using Interfaces`

```
// Interface 1
interface Parent1 {
    void fun();
}

// Interface 2
interface Parent2 {
    void fun();
}

// Class implementing both interfaces
class Test implements Parent1, Parent2 {
    // Overriding fun() method from both interfaces
    public void fun() {
        System.out.println("Child Class");
    }

    public static void main(String args[]) {
        Test t = new Test();
        t.fun();
    }
}
```

```
/*Output:

Child Class

*/
```

This demonstrates how Java allows a class to “inherit” behaviors from multiple sources via interfaces, without the ambiguity problems of multiple class inheritance.

`Example: Multiple Inheritance with Class and Interface`

You can also combine single class inheritance with interface implementation:

```java
interface Backend {
    void connectServer();
}

class Frontend {
    void responsive(String str) {
        System.out.println(str + " can also be used as frontend.");
    }
}

class Language extends Frontend implements Backend {
    String language = "Java";

    public void connectServer() {
        System.out.println(language + " can be used as backend language.");
    }

    public static void main(String[] args) {
        Language java = new Language();
        java.connectServer();
        java.responsive(java.language);
    }
}
```

```
/*Output:

Java can be used as backend language.
Java can also be used as frontend.

*/
```

Here, the Language class inherits from Frontend and also implements the Backend interface, simulating multiple inheritance.

Hybrid-Inheritance:

It's a combination of two or more types of inheritance such as single, multilevel, hierarchical, or multiple. Java does not support Hybrid inheritance with classes since it doesn't support multiple-inheritance with classes.

Hybrid inheritance is achieved by mixing class inheritance with interface implementation.

How Hybrid Inheritance Works in Java?

1. Single Inheritance : A class extends another class.
2. Hierarchical Inheritance : Multiple classes extend the same superclass.
3. Multilevel Inheritance : A class inherits from a superclass, which itself inherits from another class.
4. Multiple Inheritance (via Interfaces): A class implements multiple interfaces.

Hybrid inheritance combines these patterns, typically by having a class extend another class and also implement one or more interfaces, or by mixing multilevel and hierarchical inheritance

Ex:

```java
// Single inheritance: GrandMother → Mother → Daughter/Son (hierarchical)
class GrandMother {
    void showG() { System.out.println("She is grandmother."); }
}

class Mother extends GrandMother {
    void showM() { System.out.println("She is mother."); }
}

class Daughter extends Mother {
    void showD() { System.out.println("She is daughter."); }
}

class Son extends Mother {
    void showS() { System.out.println("He is son."); }
}

// Adding interface for multiple inheritance-like behavior
interface Father {
    void showFather();
}

class Child extends Mother implements Father {
    public void showFather() {
        System.out.println("He is father.");
    }
    void showChild() {
        System.out.println("He is child.");
    }
}

public class Main {
    public static void main(String[] args) {
        Daughter d = new Daughter();
        d.showD(); d.showM(); d.showG();

        Son s = new Son();
        s.showS(); s.showM(); s.showG();

        Child c = new Child();
        c.showFather(); c.showM(); c.showG(); c.showChild();
    }
}

/*
Output:

She is daughter.
She is mother.
She is grandmother.
He is son.
She is mother.
She is grandmother.
He is father.
She is mother.
She is grandmother.
He is child.

*/
```

This example demonstrates hybrid inheritance by combining hierarchical (GrandMother → Mother → Daughter/Son), multilevel (GrandMother → Mother → Child), and multiple inheritance-like behavior via interface implementation.

Non Access Modifiers

final:

`final` prevents modification:

- For `variables`: value cannot be reassigned.
- For `methods`: method cannot be overridden.
- For `classes`: class cannot be subclassed/extended (`inherited`).
- The keyword **`final`** marks everything non-modifiable.

  Prevents a class from being inherited or a method from being overridden or an attribute from being reassigned.

static:

It allows us to access the methods and attributes of a class without creating an object reference.

Makes the member belong to the class instead of instances:

- static variable → shared across all objects
- static method → can be called without creating an object
- static block → runs once when the class is loaded
- static class → only allowed for nested classes

Static members of a superclass can be accessed in a subclass, but:

- They are not inherited in the traditional polymorphic sense.
- They are associated with the class, not with instances.

Access Modifiers

Why?

Java access modifiers control visibility or accessibility of classes, methods, and variables across different packages and class hierarchies.

What they do?

They enforce encapsulation and define how classes, methods and attributes interact, especially in inheritance.

attributes: - Instance Vars - Local Vars - Static Vars - Parameter Vars

Encapsulation: hide internal details

1. public
2. private

3. protected
4. default

public:

Accessible everywhere.

```
The class is accessible from any other class in any package.
```

private:

Accessible only within the same class.

protected:

`protected` means that the member can be accessed by any class in the same package and by subclasses even if they are in another packages(through `inheritance` only). BUT only through the subclass reference, not through a superclass reference.

default (no need to use keyword):

Accessible within the same package.

> The class is accessible only within its own package. This is also known as "package-private".

Super/top-level class can only use public. `private` and `protected` modifiers cannot be used with Super/top-level classes. Inner classes can use all modifiers (e.g., `private`, `protected`).

Access Modifiers in the Context of Inheritance:

1. Subclass in the Same Package:

- Inherits all members except private.
- Can access public, protected, and default members directly.

Method Overriding vs Method Overloading

Method Overloading and Method Overriding are both ways to achieve polymorphism in Java.

In short:

Overloading = same method name, different parameters, same class.

Overriding = same method name and parameters, different classes (parent & child)

Method Overloading:

Multiple methods in the same class have the same name but different parameters (different types, number of params, or both).

Purpose: Allows methods to handle different types or numbers of inputs. Binding: Decided at compile time (static binding). Note: Happens within a single class only.

```java
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

Method Overriding:

When a subclass provides its own version of a parent class's method with the same name and parameters.

Example:

```java
class Animal {
    public void makeSound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

Method Overriding Rules:

Overridden methods cannot have `stricter access` than the superclass method.

> Access level must be equally or less restrictive. public > default > protected > private

```java
class Super {
    protected void show() { ... }
}

class Sub extends Super {
    @Override
    public void show() { ... }    // OK: public > protected
    // @Override
    // void show() { ... }        // Error: default < protected
}
```

Can a final method be Overloaded in Java? YES, absolutely.

Overloading depends on the method's signature (name and parameters). A final method is still a method, and you can define other methods with the same name but different parameters within the same class. The final keyword only prevents overriding, not overloading.