Java

Object oriented programming

Facts about Java

#1: Java is Platform-Independent

- Java follows the **"Write Once, Run Anywhere"** (WORA) principle.
- Java programs are compiled into **bytecode**, which runs on any platform with a **Java Virtual Machine (JVM)**.

Facts about Java

#2: Java is an Object-Oriented Programming (OOP) Language

- Java supports core **OOP principles**:
  - **Encapsulation**: Hiding data using private variables and public methods.
  - **Inheritance**: Reusing code through parent-child relationships.
  - **Polymorphism**: Method overloading and overriding for dynamic behavior.

#3: Java Supports Static and Dynamic Binding

- **Static Binding**: Occurs at compile-time (e.g., method overloading).
- **Dynamic Binding**: Occurs at runtime (e.g., method overriding).

Facts about Java

#4: Java Supports Exception Handling

- Java handles errors gracefully with try, catch, finally, and throw.
- Exceptions can be **checked** (compile-time) or **unchecked** (runtime).

#5: this Keyword

- Refers to the **current object** instance.
- Used to differentiate between instance variables and method parameters.

Facts about Java

#6: Constructor in Java

- Special method invoked when an object is created.
- Types:
  - **Default Constructor**: No parameters.
  - **Parameterized Constructor**: Accepts arguments.

#7: super Keyword

- Refers to the **parent class**.
- Used to access parent class methods, constructors, or fields.

Facts about Java

#8: instanceof Operator

- Used to check whether an object is an instance of a particular class or subclass.

#9: Polymorphism in Action

- **Compile-time polymorphism**: Method overloading.
- **Runtime polymorphism**: Method overriding.

Inhertitance

Single Inheritance Only (Classes)

- In Java, a class can **inherit from only one parent class** (single inheritance) using the extends keyword.
- Java does not support **multiple class inheritance** to avoid ambiguity.

All Classes Inherit from Object Class

- In Java, every class implicitly inherits from the **Object class**, which is the root of the class hierarchy.
- This means all classes have access to methods like toString(), equals(), and hashCode().

Inhertitance

extends Keyword for Inheritance

- The extends keyword is used for a child class to inherit from a parent class.
- It creates an **IS-A relationship** between the child and parent.

Constructors Are Not Inherited

- A subclass **does not inherit** the constructor of its parent class.
- However, the **parent class constructor** is called automatically when a child object is created (using super()).

Inhertitance

Method Overriding

- A child class can **override** a method from its parent class to provide a specific implementation.
- Overriding enables **runtime polymorphism**.

super Keyword

- The super keyword is used to:
  - **Call parent class methods**.
  - **Call parent class constructors**.
  - Access parent class fields when they are shadowed by child fields.

Inhertitance

Final Classes Cannot Be Inherited

- If a class is declared with the final keyword, it **cannot be extended**.

Final Methods Cannot Be Overridden

- A method marked with final cannot be overridden by a subclass.

Static Methods Are Not Inherited

- Static methods are **not inherited** in the same way as instance methods.
- However, they can be **hidden** by redefining them in the subclass.

Inhertitance

Protected Members Are Inherited

- protected fields and methods in a parent class are accessible in the child class.

Object Type Determines Behavior

- A parent class reference can point to a child class object. At runtime, the method of the actual object type is invoked (polymorphism).
- A private class cannot be inherited as it is not accessible outside its scope.

Static

Static Variables (Class Variables)

- A **static variable** is shared across all instances of a class.
- It is created when the class is loaded into memory and destroyed when the class is unloaded.
- All objects of the class **share the same static variable**.

Static Methods

- Static methods belong to the class, not to any specific object.
- You can call a static method using the **class name**, without creating an object.

Overriding vs Overloading

Feature	**Method Overriding**	**Method Overloading**
Definition	Redefining a parent class method in a child class.	Multiple methods with the same name but different parameters.
Class Relationship	Requires **inheritance** (parent and child classes).	Happens within the **same class**.
Access Modifier	Cannot reduce visibility (e.g., protected cannot become private).	Access modifiers can be different.
Static/Instance	Works only with **instance methods**.	Can apply to **static and instance methods**.
Runtime/Compile-Time	**Runtime polymorphism** (method resolved at runtime).	**Compile-time polymorphism** (method resolved at compile time).
Keyword Used	Uses @Override annotation (optional but recommended).	No special keyword needed.

Final keyword

- The final keyword in Java is used to declare **constants**, prevent method overriding, and restrict inheritance.
- **Final Variable**: A variable declared final **cannot be reassigned** once initialized.
- **Final Method**: A method declared final **cannot be overridden** in a subclass.
- **Final Class**: A class declared final **cannot be inherited**.

Static

Key Rules:

1. Static methods **cannot access instance variables** or instance methods directly.
2. Static methods can **only access static members** of the class.
3. this and super cannot be used inside static methods.

Static Methods Cannot Be Overridden

- Static methods **cannot be overridden** because they are resolved at **compile-time**.
- If you define a static method with the same name in a subclass, it **hides** the parent method (method hiding).

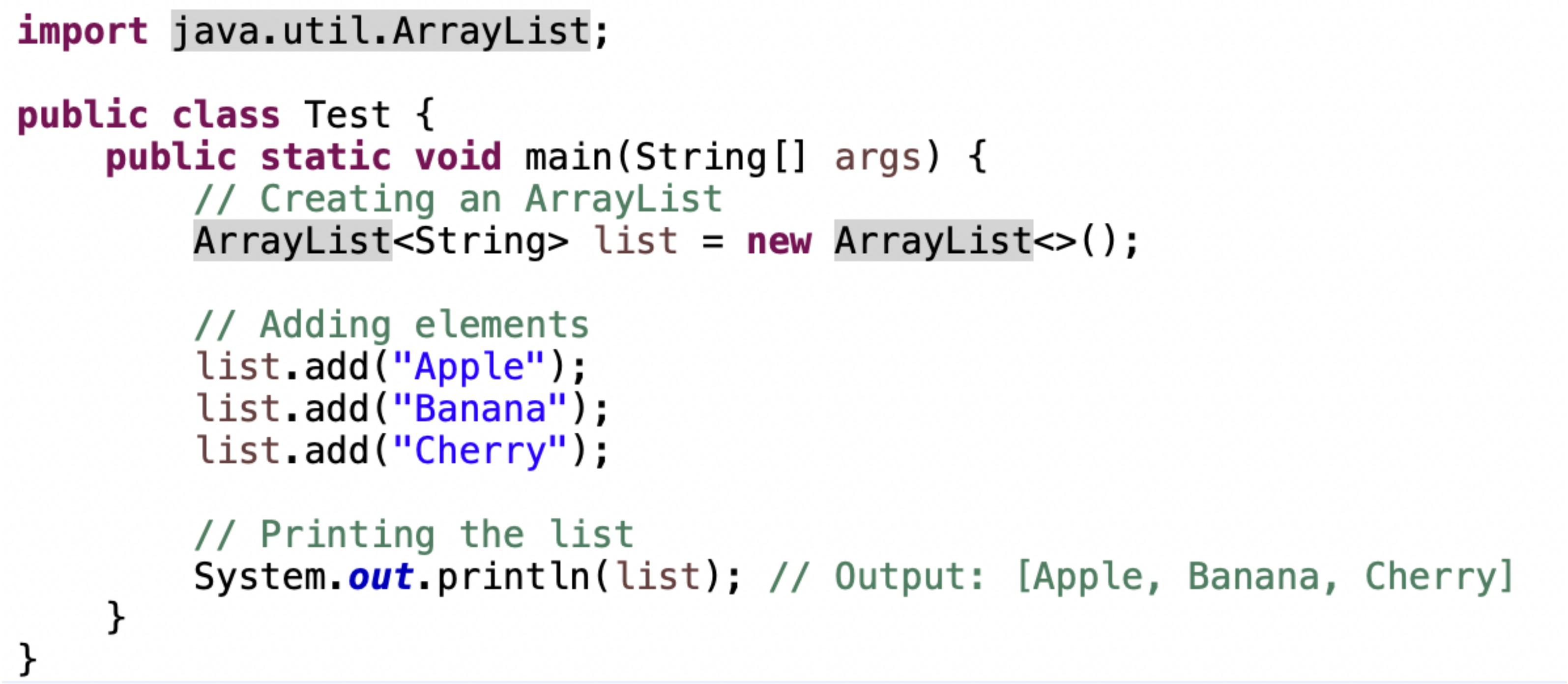
Accessibility Levels

Modifier	Same Class	Same Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

ArrayList in Java

- **ArrayList** is a **resizable array** implementation of the List interface in Java.
- It is part of the **java.util** package.
- Unlike arrays, **ArrayList** can grow and shrink dynamically as elements are added or removed.

ArrayList



```java
import java.util.ArrayList;

public class Test {
    public static void main(String[] args) {
        // Creating an ArrayList
        ArrayList<String> list = new ArrayList<>();

        // Adding elements
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        // Printing the list
        System.out.println(list); // Output: [Apple, Banana, Cherry]
    }
}
```

Characteristics of ArrayList

- **Dynamic Size**: It resizes automatically when elements are added or removed.
- **Ordered Collection**: It maintains the **insertion order** of elements.
- **Allows Duplicates**: Duplicate elements are allowed.
- **Index-Based Access**: Elements can be accessed using their **index**.
- **Non-Synchronized**: ArrayList is **not thread-safe** (use Collections.synchronizedList for thread safety).

Common Methods of ArrayList

Method	**Description**
add(E e)	Adds an element to the end of the list.
add(int index, E e)	Adds an element at a specific index.
get(int index)	Retrieves an element at a specific index.
set(int index, E e)	Replaces an element at a specific index.
remove(int index)	Removes an element at the specified index.
remove(Object o)	Removes the first occurrence of the specified element.
size()	Returns the number of elements in the list.
isEmpty()	Checks if the list is empty.
contains(Object o)	Checks if the list contains a specific element.
clear()	Removes all elements from the list.

ArrayList

-
  ```
  for (int i = 0; i < list.size(); i++)
                    { System.out.println(list.get(i)); }
  ```

-
  ```
  for (String item : list)
                   { System.out.println(item); }
  ```

Abstract

Abstract Classes and **Interfaces** are used to achieve abstraction, which allows you to define methods that must be implemented in child classes without specifying their behavior.

Abstract

An **Abstract Class** is a class that cannot be instantiated. It can contain:

- ➢Abstract methods (without implementation)
- ➢Concrete methods (with implementation)
- ➢Fields (variables)
- ➢Constructors

Abstract classes are used as **base classes** for other classes to ensure consistency and enforce certain behaviors.

Abstract

- **Cannot be instantiated**: You cannot create an object of an abstract class.
- **Can have both abstract and concrete methods**: Allows partial implementation.
- **Constructors and instance variables**: Can have constructors and can maintain state.
- **Inheritance**: A class can extend only one abstract class due to single inheritance.
- **When to Use**
- When you want to provide a common base class with default behavior.
- When subclasses share a common method implementation.
- When you need to define a template for future classes.

Abstract

1. An **abstract class** can have both **abstract** and **concrete methods**.
2. Abstract classes can have **fields** (variables) and constructors.
3. A class must **extend** an abstract class and provide implementations for all abstract methods.
4. Abstract classes are used when there is a **common base behavior** for multiple subclasses.

Abstract

- An abstract class can have **abstract methods** (methods without a body).
- Subclasses must **override** and provide implementations for all abstract methods.
- Abstract methods are declared using the abstract keyword.
- You **cannot create an object** of an abstract class directly.
- You can only create an instance of its **subclass**.
- Abstract classes can have **instance variables**, unlike interfaces. These fields can be inherited and used in subclasses.

Abstract

- Abstract classes can also have **concrete methods** (methods with implementation).This allows you to provide default behavior that can be reused or overridden in subclasses.
- Abstract classes provide **partial abstraction**. This means they can have both abstract (no implementation) and non-abstract (concrete) methods.

Interface

- An **Interface** is like a blueprint for classes. It defines a set of abstract methods (no implementation) that must be implemented by any class that **"implements"** the interface.
- **Characteristics of Interfaces:**
- Only abstract methods (prior to Java 8)
- Fields are by default public static final (constants)
- A class can **implement multiple interfaces**
- Interfaces provide full abstraction

Interface

Key Points about Interfaces:

1. All methods in an interface are **abstract** by default (before Java 8).
2. Variables in an interface are implicitly **public, static, and final**.
3. A class **implements** an interface using the implements keyword.
4. A class can implement **multiple interfaces**.
5. Interfaces allow multiple inheritance of behavior.

Interface

Full Abstraction

- Interfaces provide **100% abstraction**.
- All methods in an interface are **abstract** (without implementation) by default (prior to Java 8).

```
interface Animal
{ void sound(); // Abstract method }
```

Interface

No Constructors

- Interfaces **cannot have constructors** because they do not contain instance variables or initialization logic.
- You cannot instantiate an interface directly.
- // Not allowed: Animal a = new Animal();

Interface

Multiple Inheritance

- A class can implement **multiple interfaces**, which allows for **multiple inheritance of behavior**.
- This solves the problem of multiple class inheritance (ambiguity).
- interface Flyable { void fly(); }
- interface Swimmable { void swim(); }
- class Duck implements Flyable, Swimmable

{ public void fly() { System.out.println("Duck is flying."); }

public void swim() { System.out.println("Duck is swimming."); } }

Interface

All Methods are Public by Default

- All methods in an interface are **public** and **abstract** by default.
- You do not need to explicitly use the public keyword.
- interface Vehicle { void start(); // This is public abstract by default }

Interface

Fields are public static final

- All variables declared in an interface are **constants**.
- They are **public**, **static**, and **final** by default.
- Interfaces cannot have **instance variables**, only constants.
- interface Constants {

  int MAX_SPEED = 120; // Implicitly public static final }
- class Car implements Constants { void printSpeed() { System.out.println("Max speed is: " + MAX_SPEED); } }

Interface

- From **Java 8**, interfaces can contain: **Default Methods**: Methods with implementation (to provide backward compatibility).
- **Static Methods**: Methods that belong to the interface itself.

Interface

- A class uses the implements keyword to **implement** an interface.
- The class must provide implementations for **all abstract methods** of the interface.
- An interface can **extend another interface** using the extends keyword.

Interface

- Interfaces reduce the dependency between classes by enforcing communication through a **common contract**.
- This allows systems to be more **modular** and **loosely coupled**, making maintenance and updates easier.

Interface

Feature	**Abstract Class**	**Interface**
Methods	Can have abstract and concrete methods	Only abstract methods (prior to Java 8)
Fields	Can have instance and static variables	Only constants (public static final)
Multiple Inheritance	Supports single inheritance	Supports multiple inheritance
Access Modifiers	Can have any access modifiers	Methods are public by default
Constructors	Can have constructors	Cannot have constructors
Implementation	Subclasses extend the abstract class	Classes implement the interface

Interface

- **When to Use Abstract Class vs Interface?**
- **Use Abstract Classes** when:
  - You need to share code among several related classes.
  - You want to enforce certain behavior with default implementation.
  - You have instance variables to be shared.
- **Use Interfaces** when:
  - You need to define a contract that unrelated classes can implement.
  - You require multiple inheritance (Java does not support multiple class inheritance).
  - You need full abstraction without state or fields.