Inheritance, Packages, and Interfaces

Inheritance

- **Definition**:
  - Inheritance is a process of creating a new class (called a derived class or subclass) from an existing class (called a base class, parent class, or superclass).
  - The derived class inherits properties and methods from the base class while adding its own properties and methods.
- **Syntax**:
  - The keyword `extends` is used to create a derived class from a base class.
  - Syntax: `class SubClassName extends SuperClassName { // methods and fields }`
  - Example:

```
class Employee {
    void display() {
        System.out.println("Salary = 1000000");
    }
}

class Programmer extends Employee {
    void bonus() {
        System.out.println("Bonus = 50000");
    }
}
```

Types of Inheritance

1. **Single Inheritance**:
   - In single inheritance, a subclass is derived from a single parent class.
   - Example:

```
class Employee {
    void display() {
        System.out.println("Salary = 1000000");
    }
}

class Programmer extends Employee {
    void bonus() {
        System.out.println("Bonus = 50000");
```

```
    }
}
```

2. **Multiple Inheritance**:
    - In multiple inheritance, a subclass can have more than one parent class.
    - Java does not support multiple inheritance directly but achieves it through interfaces.
    - Example:

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

interface Bird {
    void fly();
}

class Bat implements Animal, Bird {
    public void eat() {
        System.out.println("Bat is eating");
    }

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

3. **Multilevel Inheritance**:
    - In multilevel inheritance, a class is derived from a derived class.
    - Example:

```
class Person {
    void show() {
        System.out.println("I am a person");
    }
}

class Student extends Person {
    void show() {
        System.out.println("I am a student");
    }
}

class Engineer extends Student {
    void show() {
        System.out.println("I am an engineer");
    }
}
```

4. **Hierarchical Inheritance**:

- In hierarchical inheritance, multiple derived classes inherit from a single parent class.
- Example:

```
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

class Lion extends Animal {
    void roar() {
        System.out.println("Lion roars");
    }
}

class Leopard extends Animal {
    void run() {
        System.out.println("Leopard runs fast");
    }
}
```

5. **Hybrid Inheritance**:
    - Hybrid inheritance is a combination of more than two types of inheritance (e.g., single, multiple, hierarchical).
    - This form of inheritance is not directly supported in Java.

Benefits of Inheritance

- **Software Reusability**:
    - Inheritance promotes code reuse by allowing a subclass to inherit properties and methods from a base class.
- **Increased Reliability**:
    - By reusing proven and reliable code, the software's reliability can be increased.
- **Code Sharing**:
    - Common code can be shared among different classes, reducing redundancy.
- **Consistency of Interface**:
    - Inheritance can provide consistent interfaces across different classes.
- **Rapid Prototyping**:
    - Inheritance allows developers to quickly prototype new features by extending existing classes.
- **Polymorphism and Framework**:
    - Inheritance enables polymorphism and the creation of object-oriented frameworks.
- **Information Hiding**:
    - By restricting access to certain properties and methods, information hiding can be achieved.

Costs of Inheritance

- **Execution Speed**:
  - Inheritance may slow down execution speed due to the overhead of method calls.
- **Program Size**:
  - The size of the program may increase as a result of the inheritance hierarchy.
- **Message Passing Overhead**:
  - The overhead of message passing (e.g., method calls) can impact performance.
- **Program Complexity**:
  - Inheritance can increase program complexity due to the relationships between classes.

Hierarchical Abstraction

Abstraction:

- Abstraction is the process of hiding implementation details and showing only the necessary functionality to the user.
- It is achieved using abstract classes and interfaces.

Abstract Classes:

- A class that is declared with the `abstract` keyword is known as an abstract class.
- Abstract classes contain method declarations without providing the actual implementation (i.e., abstract methods).

Hierarchical Abstraction:

- Hierarchical abstraction refers to the concept of organizing classes into a hierarchy where a superclass is extended by multiple subclasses.
- The superclass contains common attributes and behavior shared by its subclasses, while each subclass can have its own attributes and behavior.
- This hierarchical structure allows for code reuse, making the code more manageable and organized.

Example: Abstract Class and Hierarchical Structure

- **Abstract Class (`Shape`)**:

```
abstract class Shape {
    // Attributes
    int x, y;
```

```
    // Abstract method
    abstract void draw();
}
```

- **Subclass (`Circle`)**:

```
class Circle extends Shape {
    // Attributes specific to Circle
    int radius;

    // Implementing abstract method
    void draw() {
        System.out.println("Drawing a circle");
    }
}
```

- **Subclass (`Square`)**:

```
class Square extends Shape {
    // Attributes specific to Square
    int length;

    // Implementing abstract method
    void draw() {
        System.out.println("Drawing a square");
    }
}
```

- In the example above, the abstract class `Shape` serves as a superclass with an abstract method `draw()` that must be implemented by any subclass (`Circle` and `Square`).
- The `Circle` subclass extends the `Shape` class and provides its own implementation of the `draw()` method to draw a circle.
- The `Square` subclass also extends the `Shape` class and provides its own implementation of the `draw()` method to draw a square.

Forms of Inheritance

1. **Specialization**:
    - In specialization, the subclass is a more specialized version of the parent class.
    - An example can be seen in the Java hierarchy of graphical components in the AWT (Abstract Window Toolkit):
        - Components: Label, Button, TextArea, TextField, CheckBox, etc.
    - Subclasses inherit the properties and methods of the parent class but may also include additional properties and methods specific to their specialization.

2. **Specification**:
    - Specification involves defining specific methods and behaviors that should be available to subclasses.
    - In Java, this is supported through interfaces and abstract methods.
    - Example:

```
class MyActionListener implements ActionListener {
    // Body of class
}
```

    - By specifying interfaces or abstract methods, the parent class provides a blueprint for the child classes to implement.
3. **Construction**:
    - In construction, the superclass is used to provide common behavior for subclasses.
    - This type of inheritance allows for code reuse.
    - Example: A base class provides common methods that are shared by multiple subclasses.
4. **Extension**:
    - Extension occurs when a child class adds new methods and behavior to the parent class.
    - The subclass extends the functionality of the parent class without modifying existing methods.
    - Example:

```
class ExtendedClass extends BaseClass {
    // New methods and properties
}
```

5. **Limitation**:
    - Limitation occurs when the behavior of the subclass is more restrictive or limited compared to the parent class.
    - Subclasses may override methods to narrow the functionality or add additional constraints.
6. **Combination**:
    - Combination involves providing multiple inheritance and implementing multiple interfaces.
    - It allows a class to inherit from multiple sources or implement multiple interfaces to achieve a combination of behaviors.
    - Example:

```
class CombinedClass implements Interface1, Interface2 {
    // Implements methods from Interface1 and Interface2
}
```

Member Access Rules and Super Uses

Member Access Rules

1. **Access Modifiers**:
    - **Public**:
        - Members declared as `public` are accessible from any class.
        - Example:

```
class A {
    public int publicVariable = 10;
}

class B {
    public void printPublicVariable() {
        A obj = new A();
        System.out.println("Public variable: " +
obj.publicVariable);
    }
}
```

    - **Private**:
        - Members declared as `private` are accessible only within the class in which they are declared.
        - Example:

```
class A {
    private int privateVariable = 20;

    public void showPrivate() {
        System.out.println("Private variable: " +
privateVariable);
    }
}

class B {
    public void accessPrivate() {
        A obj = new A();
        // obj.privateVariable; // Error: Cannot access
private member from outside the class
        obj.showPrivate(); // Can access public method that
accesses the private variable
```

```
    }
}
```

- **Protected**:
  - Members declared as `protected` are accessible within the same package or subclasses of the class.
  - Example:

```
class A {
    protected int protectedVariable = 30;
}

class B extends A {
    public void printProtectedVariable() {
        System.out.println("Protected variable: " +
protectedVariable);
    }
}
```

- **Default (Package-private)**:
  - Members with no explicit access modifier are accessible only within the same package.
  - Example:

```
class A {
    int defaultVariable = 40; // Package-private
}

class B {
    public void printDefaultVariable() {
        A obj = new A();
        System.out.println("Default variable: " +
obj.defaultVariable);
    }
}
```

Super Keyword

The `super` keyword is used to refer to the immediate parent class of an object. It can be used in three different contexts:

1. **Accessing Parent Class Variables**:
   - When both the parent and child classes have a member with the same name, the `super` keyword can be used to access the parent class member.
   - Example:

```java
class SuperClass {
    int x = 20;
}

class SubClass extends SuperClass {
    int x = 80;

    void display() {
        System.out.println("Super class x: " + super.x);
        System.out.println("Sub class x: " + x);
    }
}

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

2. **Accessing Parent Class Methods**:
   - The `super` keyword can be used to invoke a parent class method when the subclass contains the same method (method overriding).
   - Example:

```java
class SuperClass {
    void display() {
        System.out.println("Super class method");
    }
}

class SubClass extends SuperClass {
    @Override
    void display() {
        super.display(); // Calls the parent class method
        System.out.println("Sub class method");
    }
}

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

3. **Accessing Parent Class Constructor**:
   - The `super` keyword can also be used to invoke the parent class constructor.
   - Example:

```java
class Vehicle {
    Vehicle() {
        System.out.println("Vehicle is created");
    }
}

class Bike extends Vehicle {
    Bike() {
        super(); // Invoke parent class constructor
        System.out.println("Bike is running");
    }
}

public class Main {
    public static void main(String[] args) {
        Bike bike = new Bike();
    }
}
```

Here are detailed notes on **polymorphism** and **method overriding** in object-oriented programming:

Polymorphism and Method Overriding

Polymorphism

- **Definition**:
  - Polymorphism means "many forms."
  - It refers to the ability of a function to operate differently based on the inputs or data types.
  - Polymorphism allows an object to take on many forms and behave differently depending on the context.
- **Types of Polymorphism**:
  - **Compile-Time Polymorphism (Ad Hoc or Static Polymorphism)**:
    - Achieved through method overloading and operator overloading.
    - In method overloading, methods with the same name can have different parameters.
    - Example:

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

```
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(3, 5)); // Output: 8 (int)
        System.out.println(calc.add(3.5, 2.5)); // Output: 6.0
(double)
    }
}
```

- **Run-Time Polymorphism (Pure or Dynamic Polymorphism)**:
  - Achieved through method overriding.
  - In method overriding, a subclass provides a specific implementation for a method that is already defined in its superclass.
  - This allows the subclass to define its behavior for a method in a way that is specific to the subclass.

Method Overriding

- Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.
- **Characteristics of Method Overriding**:
  - **Same Method Name**:
    - The method in the subclass has the same name as the method in the superclass.
  - **Same Parameters**:
    - The method in the subclass has the same parameters (signature) as the method in the superclass.
  - **Different Class**:
    - The method is defined in a subclass that extends the superclass.
- **Example**:

```
class Test {
    public void eat() {
        System.out.println("Hi");
    }
}

class Second extends Test {
    @Override
    public void eat() {
        System.out.println("Hello");
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Second s = new Second();
        s.eat(); // Output: Hello (overridden method)

        Test t = new Test();
        t.eat(); // Output: Hi (original method)
    }
}
```

- **Key Points**:
  - **Polymorphism**:
    - Method overriding supports runtime polymorphism, allowing methods to be invoked based on the actual object's type.
  - **Dynamic Binding**:
    - Method overriding relies on dynamic binding (late binding), where the method to be called is determined at runtime based on the object's type.
  - **Access Level**:
    - In method overriding, the subclass method should have the same or broader access level (e.g., public, protected) as the superclass method.
  - **Return Type**:
    - The return type of the subclass method must be the same as, or a subtype of, the return type of the superclass method.
  - **Use with Inheritance**:
    - Method overriding occurs in the context of inheritance, where a subclass extends a superclass.
  - **Annotations**:
    - The `@Override` annotation can be used in Java to explicitly indicate that a method in a subclass is intended to override a method in the superclass.
    - This helps prevent errors due to typos or mismatches in method signatures.
- **Example**:

```
class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

class Bike extends Vehicle {
    @Override
    void run() {
        System.out.println("Bike is riding");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Bike bike = new Bike();
        bike.run(); // Output: Bike is riding
    }
}
```

Abstract Classes and the Object Class

Abstract Classes

- **Definition**:
  - An abstract class is a class that contains one or more abstract methods (methods without implementation) and is declared with the `abstract` keyword.
  - Objects cannot be created directly from an abstract class; it must be subclassed.
  - Abstract classes can also contain non-abstract methods with full implementations.
- **Abstract Methods**:
  - An abstract method is a method that is declared with the `abstract` keyword.
  - It does not contain a method body; it is a method signature followed by a semicolon.
  - Abstract methods define actions that must be implemented by subclasses.
- **Example**:

```
// Abstract class example
abstract class Fruits {
    // Abstract method
    abstract public void taste();
}

// Subclass that implements the abstract method
class Apple extends Fruits {
    @Override
    public void taste() {
        System.out.println("Apple tastes sweet and crisp.");
    }
}

class Main {
    public static void main(String[] args) {
        Fruits apple = new Apple();
        apple.taste(); // Output: Apple tastes sweet and crisp.
    }
}
```

- Subclasses must implement all abstract methods from the superclass; otherwise, they themselves must be declared as abstract.
- Abstract classes allow you to define common behavior and methods that subclasses can override or use as is.
- An abstract class can have both abstract and non-abstract methods, and may also contain fields and constructors.

The Object Class

- **Definition**:
  - The `Object` class is the root class from which all classes in Java are derived.
  - It serves as the base class for all other classes in Java.
  - All classes in Java automatically inherit from the `Object` class, even if they do not explicitly extend another class.
- **Common Methods**:
  - The `Object` class provides several common methods that are inherited by all classes, such as:
    - `toString()`: Returns a string representation of the object.
    - `equals(Object obj)`: Checks whether this object is equal to another object.
    - `hashCode()`: Returns a hash code value for the object.
    - `getClass()`: Returns the runtime class of an object.
    - `clone()`: Creates a copy of the object (if the class implements the `Cloneable` interface).
    - `finalize()`: Invoked by the garbage collector before an object is destroyed.
    - `wait()`, `notify()`, `notifyAll()`: Used for thread synchronization.
- **Example**:

```
// Demonstrating methods from the Object class
class Animal {
    // Method from Object class
    public void makeSound() {
        System.out.println("Animal is making a sound");
    }
}

class Dog extends Animal {
    // Overriding method from Animal class
    @Override
    public void makeSound() {
        System.out.println("Dog is barking");
    }
}

class Main {
```

```
    public static void main(String[] args) {
        // Creating objects
        Animal animal = new Animal();
        Dog dog = new Dog();

        // Calling methods
        animal.makeSound(); // Output: Animal is making a sound
        dog.makeSound(); // Output: Dog is barking
    }
}
```

- Since all classes in Java inherit from `Object`, they all inherit the common methods it provides.
- Overriding methods such as `toString()` and `equals()` can help provide more meaningful behavior for your classes.
- Classes that override `clone()` must implement the `Cloneable` interface.

Defining, Creating, and Accessing a Package

Packages are a way to organize related classes, interfaces, and sub-packages in Java. They help manage the codebase more efficiently and avoid naming conflicts.

Package Definition

- **Definition**:
  - A package is a container for a group of related classes and interfaces.
  - The package name and the directory name should be the same.
  - Packages promote reusability and encapsulation.
  - They also help in categorizing classes and interfaces for easier maintenance.
- **Advantages**:
  - **Name conflict resolution**: Packages prevent naming conflicts by organizing classes with similar names into separate packages.
  - **Access control**: Packages allow you to specify access levels for classes and interfaces.
  - **Data encapsulation**: Packages enable data encapsulation, providing better security and privacy.
  - **Maintenance**: Organizing code into packages makes it easier to maintain and navigate the codebase.

Types of Packages

- **User-Defined Packages**:
  - These are packages created by the user to organize their classes and interfaces.

    - You can create a user-defined package by declaring a package at the beginning of the Java source file.
- **Built-In Packages**:
    - These are predefined packages that are part of the Java Development Kit (JDK).
    - Examples include `java.util`, `java.io`, `java.lang`, and more.
    - Built-in packages provide a wide variety of classes and interfaces for common tasks.

Creating a Package

- **In an Integrated Development Environment (IDE)**:
    - IDEs like Eclipse allow you to create packages easily.
    - Define the package name at the beginning of the source file using the `package` keyword.
- **In a source file**:
    - Declare the package name at the top of your Java source file using the `package` keyword.
    - The package name should match the directory structure.
    - Example:

```
package mypackage;

public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello from package!");
    }
}
```

- **Compiling**:
    - Use the `-d` flag with the `javac` command to specify the output directory for compiled classes.
    - Example:
        - `javac -d bin MyClass.java` to compile a class and place the class file in the `bin` directory.
- **Running**:
    - Run the compiled class using the package name and the class name.
    - Example:
        - `java mypackage.MyClass` to run the `MyClass` class in the `mypackage` package.

Accessing a Package from Another Package

- **Import Statements**:

- Use the `import` keyword to import classes and interfaces from other packages.
- `import packageName.*`: Imports all classes and interfaces from the specified package (but not sub-packages).
- `import packageName.ClassName`: Imports a specific class from the specified package.

- **Fully Qualified Names**:
  - You can also use the fully qualified name of the class (package name + class name) to access classes and interfaces from other packages.
- **Example**:

```
// File: mypackage/Apple.java
package mypackage;

public class Apple {
    public void show() {
        System.out.println("Apple from mypackage.");
    }
}

// File: myotherpackage/Banana.java
package myotherpackage;

import mypackage.Apple;

public class Banana {
    public static void main(String[] args) {
        Apple apple = new Apple();
        apple.show(); // Output: Apple from mypackage.
    }
}
```

Understanding CLASSPATH and Importing Packages

`PATH`

- **Definition**:
  - `PATH` is an environment variable that tells the operating system where to find the executable files (e.g., `javac`, `java`) necessary for running Java programs.
  - If the Java compiler (`javac`) and Java interpreter (`java`) are not in the system's `PATH`, the system will not be able to find them.
- **Setting `PATH`**:
  - **Temporary**:
    - Open the command prompt and set the `PATH` variable temporarily for the current session.

        - Example: `set PATH=C:\path\to\classes;%PATH%`
    - **Permanent**:
        - Go to **Control Panel** > **System and Security** > **System** > **Advanced system settings**.
        - In the **System Properties** window, click on **Environment Variables**.
        - Under **System Variables** or **User Variables**, select `PATH` and add the path to the Java bin directory.

`CLASSPATH`

- **Definition**:
    - `CLASSPATH` is an environment variable that tells the Java Virtual Machine (JVM) and Java compiler where to find the class files (`.class` files).
    - It is used by the class loader to locate class files and load them during runtime.
- **Class Loader**:
    - **Function**:
        - The class loader is responsible for dynamically loading Java classes into the JVM during runtime.
        - It searches for classes based on the `CLASSPATH`.
    - **Types**:
        - **Bootstrap Class Loader**: Loads JDK internal classes.
        - **Extension Class Loader**: Loads JDK extension directory classes.
        - **System Class Loader**: Loads classes from the current `CLASSPATH`.
        - **Custom Class Loaders**: User-defined class loaders.
- **When to Set `CLASSPATH`**:
    - If Java source files and class files are stored in different locations, you may need to set the `CLASSPATH` variable to enable the program to find the required classes.
- **Setting `CLASSPATH`**:
    - **Temporary**:
        - Open the command prompt and set the `CLASSPATH` variable temporarily for the current session.
        - Example: `set CLASSPATH=C:\path\to\classes;%CLASSPATH%`
    - **Permanent**:
        - Go to **Control Panel** > **System and Security** > **System** > **Advanced system settings**.
        - In the **System Properties** window, click on **Environment Variables**.
        - Under **System Variables** or **User Variables**, add the paths to class files in the `CLASSPATH` variable.

Importing Packages

- **Import Statements**:
  - In Java, you can use the `import` keyword to import classes and interfaces from other packages.
  - This makes the classes and interfaces from the specified package available to your program.
  - Example:
    - `import java.util.ArrayList;` imports the `ArrayList` class from the `java.util` package.
  - **Wildcard Import**:
    - Using `import packageName.*` imports all classes and interfaces from the specified package (but not from sub-packages).
    - Example: `import java.util.*;` imports all classes from the `java.util` package.

Differences Between Classes and Interfaces

Aspect	Class	Interface
Definition	A blueprint for creating objects.	A contract specifying methods and fields.
Implementation	Instantiated to create objects.	Cannot be instantiated directly.
Declaration	Declared using the `class` keyword.	Declared using the `interface` keyword.
Methods	Can have concrete, abstract, static, and final methods.	Only abstract methods, unless specified as default or static.
Variables	Can have instance, class, and local variables with different access modifiers.	Only constants (`public` , `static` , and `final` by default).
Access Modifiers	Methods and variables can have various access modifiers (e.g., `public` , `private` , `protected`).	Methods are `public` by default; variables are `public` , `static` , and `final` .
Inheritance	Supports single inheritance (`extends` one superclass).	Supports multiple inheritance (can implement multiple interfaces).
Extending	Extends another class using the `extends` keyword.	Can extend other interfaces using the `extends` keyword.
Implementing	Implements an interface using the `implements` keyword.	Cannot implement other classes or interfaces.
Use Cases	Defines objects and their behaviors.	Defines a contract for classes to implement specific methods and fields.

Aspect	Class	Interface
Abstraction	Can provide partial abstraction (with abstract methods).	Used primarily for achieving full abstraction.
Usage in Program Structure	Used to define objects and encapsulate code.	Provides a way to separate interface from implementation.
Multiple Inheritance	Not supported in classes directly.	Supported, as a class can implement multiple interfaces.

Defining and Implementing Interfaces

Defining Interfaces:

- **Keyword**: Interfaces are declared using the `interface` keyword.
- **Purpose**: Interfaces are used to achieve data abstraction by defining a contract or blueprint for classes to implement.
- **Structure**: An interface consists of abstract methods and may include constant variables, static methods, and default methods.
- **Abstraction**: Interfaces are similar to classes but do not contain method bodies (unless they are default or static methods).
- **Multiple Inheritance**: Interfaces can be used to support multiple inheritance in Java.
- **Syntax**:

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

Implementing Interfaces:

- **Usage**: To use an interface in a class, use the `implements` keyword followed by the interface name.
- **Method Implementation**: When a class implements an interface, it must provide concrete implementations for all the abstract methods defined in the interface.
- **Example**:

```
public class Mammals implements Animal {
    public void eat() {
        System.out.println("Mammals eat food");
    }

    public void move() {
        System.out.println("Mammals move around");
```

```
    }
}
```

Extending Interfaces:

- **Inheritance**: Interfaces can extend other interfaces using the `extends` keyword.
- **Multiple Inheritance**: An interface can extend multiple interfaces, allowing for a form of multiple inheritance.
- **Example**:

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

interface B extends A {
    void methodB();
}
```

Nested Interfaces:

- **Inner Interface**: An interface can be defined within another interface or class, referred to as a nested interface or inner interface.
- **Visibility**: Nested interfaces are subject to the same visibility rules as classes; they can be `public`, `protected`, or `private`.

Variables in Interfaces:

- **Constant Variables**: Variables in interfaces are implicitly `public`, `static`, and `final` (constants).
- **Example**:

```
interface Constants {
    int MAX_VALUE = 100;  // public static final int MAX_VALUE = 100;
}
```

Applications of Interfaces:

1. **Abstraction**: Interfaces allow you to define an abstract blueprint that classes must follow.
2. **Multiple Inheritance**: Interfaces support multiple inheritance, enabling a class to implement multiple interfaces and thus inherit from multiple sources.
3. **Encapsulation**: By using interfaces, you can encapsulate related behaviors and ensure consistency across classes.

4. **Design Patterns**: Interfaces are commonly used in design patterns such as the Factory, Strategy, and Observer patterns.

Applying Interfaces and Extending Interfaces

Applying Interfaces:

- **Implementation**: A class applies an interface by using the `implements` keyword followed by the interface name.
- **Method Implementation**: When a class implements an interface, it must provide concrete implementations for all the abstract methods defined in the interface.
- **Benefits**:
  - **Abstraction**: Interfaces allow you to define an abstract blueprint for classes to implement.
  - **Polymorphism**: Implementing an interface allows classes to be treated as instances of the interface, supporting polymorphism.
  - **Flexibility**: Interfaces enable flexibility in code, as different classes can implement the same interface in different ways.
- **Example**:

```
public class Mammals implements Animal {
    public void eat() {
        System.out.println("Mammals eat food");
    }

    public void move() {
        System.out.println("Mammals move around");
    }
}
```

Extending Interfaces:

- **Inheritance**: Interfaces can extend other interfaces using the `extends` keyword.
- **Multiple Inheritance**: An interface can extend multiple interfaces, allowing for a form of multiple inheritance.
  - This supports flexibility and allows the interface to inherit the abstract methods of other interfaces.
- **Example**:

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

```
interface B extends A {
    void methodB();
}
```

- **Nested Interfaces**:
  - An interface can be defined within another interface or class, known as a nested interface or inner interface.
  - Nested interfaces follow the same visibility rules as classes (e.g., public, protected, private).
  - **Example**:

```
class Outer {
    interface Inner {
        void innerMethod();
    }
}
```

Using Interfaces in Practice:

- **Design Patterns**: Interfaces are often used in design patterns such as Factory, Strategy, and Observer.
- **Stack Operations**: Interfaces can be used to define stack operations (e.g., push and pop).
- **Example**:

```
interface Stack {
    void push(int item);
    int pop();
}

public class ArrayStack implements Stack {
    // Implementation of stack operations
    public void push(int item) {
        // Push item onto stack
    }

    public int pop() {
        // Pop item from stack
        return 0; // Example return value
    }
}
```

Variables in Interfaces and Exploring java.io

Variables in Interfaces:

- **Constant Variables**:
  - In an interface, all variables are implicitly `public`, `static`, and `final`.
  - This means the variables in interfaces act as constants; they cannot be modified once initialized.
  - Example:

```
interface Constants {
    int MAX_VALUE = 100;  // This is a constant
}
```

  - Variables declared in interfaces are accessible from any class that implements the interface, using the interface's name as a qualifier (e.g., `Constants.MAX_VALUE`).
  - Since the variables are `static`, they are associated with the interface itself, not with any implementing class.

Exploring `java.io` Package:

```
- The `java.io` package provides classes for handling input and output operations in Java, such as reading from and writing to files, and working with streams.
- The package contains various interfaces and classes to manage data flow in different formats and sources.
```

- **Common Interfaces**:
  - `java.io.Closeable`: An interface that declares a `close()` method for closing streams or resources.
  - `java.io.DataInput` and `java.io.DataOutput`: Interfaces for reading from and writing to data streams.
- **Common Classes**:
  - `java.io.File`: Represents files and directories, providing methods to manipulate file attributes and perform file I/O operations.
  - `java.io.FileInputStream` and `java.io.FileOutputStream`: Classes for reading from and writing to files using byte streams.
  - `java.io.InputStream` and `java.io.OutputStream`: Abstract classes for working with streams in general, such as file or network streams.
  - `java.io.Reader` and `java.io.Writer`: Abstract classes for handling character-based input and output operations.
- **Working with Files**:
  - You can use classes like `FileInputStream` and `FileOutputStream` to read from and write to files.

- Classes like `BufferedReader` and `BufferedWriter` provide efficient reading and writing operations by buffering data.
- Example:

```
import java.io.*;

public class FileExample {
    public static void main(String[] args) throws IOException {
        // Writing to a file
        FileOutputStream fileOutputStream = new
FileOutputStream("output.txt");
        fileOutputStream.write("Hello, world!".getBytes());
        fileOutputStream.close();

        // Reading from a file
        FileInputStream fileInputStream = new
FileInputStream("output.txt");
        int content;
        while ((content = fileInputStream.read()) != -1) {
            System.out.print((char) content);
        }
        fileInputStream.close();
    }
}
```
