Java Part 7: Inheritance, Interfaces and Polymorphism

Business concepts come in families: many kinds of claims, accounts or products that share a core and differ in details. Java lets classes share behaviour without copying code. Four steps cover it: extend, override, agree on a contract and substitute. In this part, Yousef, a claims analyst at an insurance company, uses them to process motor, property and medical claims through one piece of code, and each step answers one practical question.

1. Extend

Inheritance lets a new class, the subclass, start from an existing class, the superclass, using the keyword extends. The subclass receives the fields and methods that the superclass allows it to see, and adds its own. Private members stay hidden inside the superclass, which is another reason to give them getters. A class can extend only one other class, and every class ultimately extends Object, which is where toString comes from. A subclass constructor normally starts by calling the superclass constructor with the word super, so the shared data is built first. Inheritance fits a real is-a relationship: a motor claim is a kind of claim. The question for this step is: what do these kinds of thing truly share?

Example: A bank has savings accounts and current accounts. Both have an owner and a balance, so a developer puts these two fields in one Account class. A savings account class then extends Account and adds only an interest rate, while a current account class adds an overdraft limit. Each kind reuses the shared owner and balance and adds only what makes it different.

At work: Yousef finds three classes for motor, property and medical claims, each with the same claim number, claimant and date. He moves these shared fields and their getters into a Claim class. MotorClaim, PropertyClaim and MedicalClaim now extend Claim and add only their own details, such as a vehicle or a diagnosis code. Each constructor starts with a call to super that passes the shared data up.

2. Override

A subclass can replace a method it inherits with its own version, which is called overriding. The new method has the same name and parameters, and we mark it with the annotation Override so the compiler checks that we really are replacing something. Inside the new method, the word super followed by the method name calls the parent version, which is useful when we only want to add to the parent behaviour. When the method is called, Java runs the version that belongs to the actual object, even if the variable has the parent type. A method marked final cannot be overridden, and a class marked final cannot be extended. Overriding differs from overloading, because the parameters stay the same and the family of classes decides which version runs. The question for this step is: what does this kind do differently?

Example: A shipping company has a method that calculates the fee for any parcel from its weight. For fragile parcels, the subclass overrides this method, calls the parent version first and adds a handling charge on top. A normal parcel still uses the parent version unchanged. The annotation Override would also warn about a spelling mistake in the method name.

At work: Every claim has a payout method, but each kind calculates it differently. MotorClaim subtracts the deductible from the repair cost, and MedicalClaim applies the coverage percentage to the invoice. Yousef marks each version with Override, and a typo in a method name is caught by the compiler at once. PropertyClaim calls the basic payout of the parent first and then adds the contents allowance.

3. Contract

An interface is a contract: it lists methods that a class promises to provide, without saying how. A class signs the contract with the keyword implements, and the compiler checks that every listed method exists. A class

This document is a property of Asaseeyat Academy - © Copy Right Protected.

can extend only one class but can implement many interfaces, which makes interfaces the flexible tool. An abstract class sits between the two: it cannot be instantiated, and it may declare abstract methods that every subclass must implement, while sharing code for the rest. Interfaces describe what something can do, while inheritance describes what something is. Small interfaces with one clear purpose are easier to implement and to replace. The question for this step is: what must every kind promise to do, regardless of how?

Example: A payment system defines a contract named PaymentMethod with one method that charges an amount. Cards, bank transfers and wallets each implement the contract in their own way, so a card contacts a bank while a wallet reduces a stored balance. The checkout code only knows the contract, so it asks any payment method to charge the amount. A new way to pay joins by signing the contract, and the checkout code stays as it is.

At work: Yousef extracts the payout rule into a contract: `interface Payable { BigDecimal payout(); }`. The Claim class implements Payable, so every claim type must provide a payout method. Later, the finance team's refund class implements it too, although it is not a claim. The compiler refuses any class that signs the contract and forgets the method.

4. Substitute

Polymorphism means many forms: a variable of a parent type or of an interface type can hold any object that fits it. We write code against the general type, and at run time Java picks the version of the method that belongs to the actual object. So one loop over a list of claims can total the payouts of motor, property and medical claims without knowing which is which. When a new kind appears, we add a class and the existing code keeps working. Sometimes we must ask what an object really is, with instanceof, but needing it often is a warning sign. The question for this step is: can the calling code work with the general type only?

Example: A video platform plays files of many formats through one Player interface. The screen code calls play on a Player and never checks which format the file has. Each format class implements play in its own way, and Java runs the right one when the button is pressed. Adding a new format means adding one class, and the screen code stays the same.

At work: Yousef keeps all open claims in a list of Payable and loops over it with a for each loop. The loop calls payout on each element, and Java chooses the right calculation for each object. When the company launches a cyber insurance claim, he adds one class and changes no line of the loop. The month-end report totals every payout in one pass.

Conclusion

To recap, inheritance lets a subclass reuse a superclass, and overriding lets it do one thing differently. Interfaces describe what a class promises to do, and abstract classes mix shared code with required methods. Polymorphism lets code work with the general type while Java picks the right version at run time. Yousef's month-end report now handles every claim type through one loop, and new types plug in without touching it. Next time, we will deal with things that go wrong: exceptions, and reading files. A good habit is to prefer small interfaces and to use inheritance only for a true is-a relationship.

Sources

- Oracle: The Java Tutorials, Inheritance – https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
- Oracle: The Java Tutorials, Interfaces – https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
- Oracle: The Java Tutorials, Polymorphism – https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html
- dev.java: Learn Java – https://dev.java/learn/
- Java is a trademark of Oracle and/or its affiliates. This video is independent training material and is not affiliated with Oracle.
- Reviewed 29 Oct against Oracle tutorials: subclasses.html, override.html, createinterface.html, polymorphism.html, abstract.html, final.html, multipleinheritance.html

This document is a property of Asaseeyat Academy - © Copy Right Protected.