GUVI | HCL
Skill Up. Level Up

Core OOP: Polymorphism,Abstraction, & More!

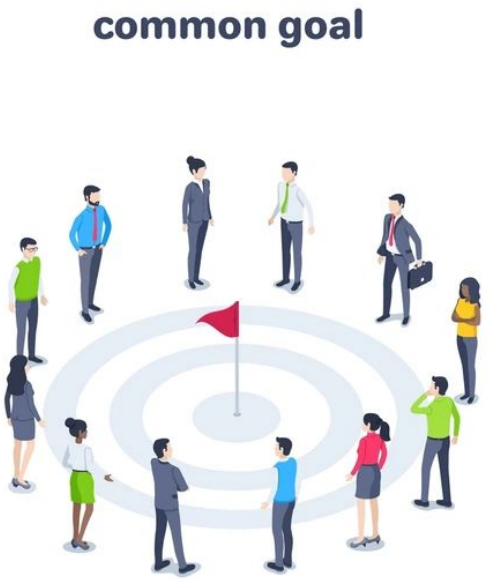
Programming Foundation

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

- What you will learn today
    - Subclassing using Polymorphism
    - Abstraction
    - Creating contracts Interfaces
    - Imports & packages
- How this builds on previous lectures
    - Classes can now be a parent or child
    - Methods in a class may now have a signature to which they must adhere



Polymorphism

From methods to behaviors

- We move from just calling functions to swapping behaviors without changing callers
    - Behavior is what an object does when you call a method on it
- Same call can trigger different behavior at run time. Eg:
    - Payments: cash, debit, credit, etc
    - Notifications: SMS, WhatsApp, etc
    - Ticket class changes while the booking flow stays stable
    - Notifications can send by SMS, WhatsApp, or email

Polymorphism: one call, many behaviors

- Polymorphism means one parent type can refer to many child types
    - The call is the same but the behavior depends on the actual object
    - Eg: Notification -> WhatsAppNotification and SMSNotification
- Objects provide their own method bodies
    - Eg: Notification::sendNotification
        - WhatsAppNotification::sendNotification
        - SMSNotification::sendNotification
- Runtime dispatch chooses which overridden method runs
    - Eg: notification.sendNotification(); OR whatsAppNotification.sendNotification();
- Storage and loops
    - Use an array of the parent type to hold different child objects
- Boundaries: Methods, not fields, are polymorphic

Overriding vs Overloading

- Overriding
  - Subclass replaces a parent method with the same signature
  - Uses @Override to let the compiler verify
- Overloading
  - Same class name with different parameter lists
  - Overloading is not polymorphism
- Key rules for overriding
  - Access cannot be stricter than the parent
  - Return can be a subtype (the child)
  - Keep behavior well defined for boundary inputs

```
class TrainTicket {                    // parent
  int fare(int km) { return (km <= 0) ? 0 : km * 2; }
}

class AcTicket extends TrainTicket { // overriding
  @Override public int fare(int km) {
    return (km <= 0) ? 0 : km * 3;
  }
}

class FlexCalc {                       // overloading in same class
  int price(int km) { return (km <= 0) ? 0 : km * 3; }
  int price(int km, int surcharge) { return price(km) +
surcharge; }
}
// Polymorphism: TrainTicket t = new AcTicket(); t.fare(10) →
30
// Overloading: new FlexCalc().price(10, 5) → 35
```

Train tickets: what varies vs what stays

- Stable data
  - PNR, passenger name, travel date
- Varying behavior
  - fare(km) calculation
  - coachType() label
- Design direction
  - Put shared data in a base class
  - Let children supply their own fare(km) and optionally coachType()

```
class TrainTicket {
  // shared default
  String coachType() { return "Sleeper"; }
  int fare(int km) { return (km <= 0) ? 0 : km * 2; } // boundary
}
class AcTicket extends TrainTicket {
  @Override String coachType() { return "AC"; }
  @Override int fare(int km) { return (km <= 0) ? 0 : km * 3; }
}
```

Interfaces

Abstract classes: declare the contract

- Abstract class
  - Cannot be created directly; may have fields, constructors, concrete and abstract methods
- Abstract method
  - Declares what children must implement (no body here)
- Relevance in the train example
  - Enforce that every ticket defines fare(km)

```
abstract class TrainTicket {
  final String pnr;
  TrainTicket(String p) { this.pnr = p; }
  abstract int fare(int km);                // must implement
  String coachType() { return "Sleeper"; }   // optional
override
}
class AcTicket extends TrainTicket {
  AcTicket(String p) { super(p); }
  @Override int fare(int km) { return (km <= 0) ? 0 : km *
3; }
  @Override String coachType() { return "AC"; }
}
```

Overriding rules that matter

- Signature match
  - Same name and parameters; @Override verifies the match
- Access
  - Child access is the same or more open than parent
- Return
  - May be a subtype of the parent's return (covariant return)
- Final
  - A final method cannot be overridden
  - Applies to variables - a variable declared with final cannot have it's value changed

```
class Base {
  protected Number price() { return 10; }
  public final int code() { return 42; } // non-overridable
}
class Child extends Base {
  // more open + subtype return
  @Override public Integer price() { return 12; }
  // @Override int code() { return 7; } // not allowed (final in Base)
}
```

@Override: make the compiler your guard

- What is it?
  - Annotation that tells the compiler “this method must override a parent method”
- Why use it?
  - Catches typos and wrong parameter types that would create an overload
- Rule of thumb
  - Put @Override on every intended override

```
class TrainTicket {
  int fare(int km) { return (km <= 0) ? 0 : km * 2; }
}

class AcTicket extends TrainTicket {
  // typo without @Override would silently overload: 'kms' ≠ 'km'
  @Override int fare(int km) { return (km <= 0) ? 0 : km * 3; }
}
```

Activity 1: Train tickets

- Goal
    - Build TrainTicket base; SleeperTicket, AcTicket override fare(int km)
- Design / Rules
    - fare(km) is abstract or overridable in base
    - Boundary rule: km <= 0 -> 0 in every child
    - Optional coachType() label differs per child
- Acceptance checks
    - new SleeperTicket().fare(10) -> 20
    - new AcTicket().fare(10) -> 30
    - TrainTicket[] arr = { new SleeperTicket(), new AcTicket() }
    - Loop prints two fares with no if on class type

Activity 1: Solution recap

- Class shape
  - Base holds shared pieces
  - children implement fare(km) and optionally coachType()
- Proved polymorphism
  - Same call t.fare(10) produced different results in a loop
- Boundary confirmed
  - km <= 0 -> 0 for all ticket types
- Boundary rules
  - perKmFare below one becomes one
  - startKm below zero becomes zero

```
TrainTicket[] arr = { new SleeperTicket(), new AcTicket() };
for (int i = 0; i < arr.length; i++) {
  System.out.println(arr[i].fare(10));  // 20, 30
}
```

Interfaces

Interfaces: contracts without shared state

- What is an interface?
  - A type that declares method signatures; implementing classes must provide bodies
  - No shared object state in the interface
- Use case
  - Many classes share a capability, not shared data
  - Eg: Plug in new payment providers without touching callers

```
// contract
interface Payable { int pay(int amount); }

class CreditCard implements Payable {
  @Override public int pay(int amount) {
    System.out.println("Credit card paid " + amount);
    return amount;
  }
}
class DebitCard implements Payable {
  @Override public int pay(int amount) {
    System.out.println("Debit card paid " + amount);
    return amount;
  }
}
```

Default & static methods on interfaces

- Default method
  - A reusable body shared by implementers (good for small helpers)
  - Use Case: To add new functionality to interfaces without breaking existing classes
- Static method
  - Utility on the interface type
  - It can be called using the interface name. Eg: InterfaceName.method()
- Use cases & benefits
  - Defaults reduce duplication; static gives a central utility without a class

```
interface Payable {
  int pay(int amount);
  default int validate(int amount) { return (amount <= 0) ?
0 : amount; }
  static String currency() { return "INR"; }
}

class Upi implements Payable {
  @Override public int pay(int amount) {
    int a = validate(amount);
    System.out.println("UPI " + a + " " +
Payable.currency());
    return a;
  }
}
```

Multiple interfaces: combining capabilities

- One class, many capabilities
  - Implement several small interfaces in one class
- What's combined?
  - Method obligations and default bodies, not fields or state
- Use cases & benefits
  - Payments that can also refund; orthogonal features stay decoupled

```
interface Payable { int pay(int amount); }
interface Refundable { int refund(int amount); }

class CardGateway implements Payable, Refundable {
 @Override public int pay(int amount)    {
         System.out.println("Card pay " + amount);
         return amount;
 }
 @Override public int refund(int amount) {
         System.out.println("Card refund " + amount);
         return amount;
 }
}
class App {
 public static void main(String[] args) {
   Payable p = new CardGateway(); p.pay(300);
   Refundable r = new CardGateway(); r.refund(100);
 }
}
```

Resolving default-method conflicts

- When conflict happens
  - Two interfaces provide the same default method name and signature
- Resolution rule
  - Override in the class and select explicitly: InterfaceName.super.method()
- Benefit
  - You stay explicit about which behavior you keep or combine

```
interface A { default String note(){ return "A"; } }
interface B { default String note(){ return "B"; } }

class Combo implements A, B {
  @Override public String note() {
    return A.super.note() + "+" + B.super.note(); //
choose/compose
  }
}

class App {
  public static void main(String[] args) {
    System.out.println(new Combo().note()); // A+B
  }
}
```

Activity 2: Card payments

- Goal
    - Define Payable and implement CreditCard, DebitCard, Upi
- Design / Rules
    - default int validate(int amount) normalizes input
    - pay(amount) returns the validated amount and prints provider line
- Acceptance checks
    - new CreditCard().pay(500) -> 500 with provider print
    - new DebitCard().pay(0) -> 0 using validate
    - Payable[] arr = { new CreditCard(), new Upi() } loop calls pay(300) with two provider-specific prints
    - No if on class type in the driver

Activity 2: Solution recap

- Interface + classes
  - Payable with validate (default) and currency (static)
  - CreditCard, DebitCard, Upi override pay
- Polymorphism
  - Same call p.pay(…) yields provider-specific prints in a loop
  - pay(amount) returns the validated amount and prints provider line
- Boundary
  - Amount <= 0 -> 0 via validate

```
Payable[] arr = { new CreditCard(), new DebitCard(), new
Upi() };
for (int i = 0; i < arr.length; i++) {
  int out = arr[i].pay((i == 1) ? 0 : 300);
  System.out.println("out -> " + out);
  // Expected prints: CC line, Debit line (0), UPI line
}
```

Polymorphism w/ interfaces

- Polymorphism can be used with interfaces in addition to classes
- A single call like notifier.send(msg) on a common type (interface or base class) dispatches to the actual object
- From behavior swapping to channels
    - One call send(msg); different notifier objects handle it (SMS, WhatsApp, Email)
- Why this matters
    - Switch channels without touching calling code

```
interface Notifier { void send(String msg); }

class SmsNotifier implements Notifier {
  @Override public void send(String msg){
System.out.println("SMS -> " + msg); }
}
class WhatsAppNotifier implements Notifier {
  @Override public void send(String msg){
System.out.println("WA -> " + msg); }
}

class App {
  public static void main(String[] args) {
    Notifier n = new SmsNotifier();  n.send("Order packed");
    n = new WhatsAppNotifier();      n.send("Out for delivery");
  }
}
```

Packages

Packages: organizing rail, payments, notify

- Map classes to folders; eg: com.rail.*, com.payments.*, com.notify.*
- Top line must match directory; eg: package com.rail; for files under com/rail
- Import across packages; use explicit imports for clarity

```
// file: com/rail/AcTicket.java
package in.rail;
public class AcTicket { public int fare(int km){ return (km <= 0) ? 0 : km * 3; } }

// file: com/payments/CreditCard.java
package in.payments;
public class CreditCard { public int pay(int amount){ System.out.println("CC -> " + amount); return amount; } }

// file: App.java
import in.rail.AcTicket;
import in.payments.CreditCard;
class App {
  public static void main(String[] args) {
    System.out.println(new AcTicket().fare(10)); // 30
    new CreditCard().pay(500);                   // CC -> 500
  }
}
```

Imports and name clashes

- Same simple name in two packages
    - Import one; use fully qualified name (FQCN) for the other
- Be explicit
    - Avoid wildcard imports when names collide

```
// Assume both classes exist with simple name 'Ticket'
import com.rail.Ticket;                 // main use in this file

class App {
  public static void main(String[] args) {
    Ticket rail = new Ticket();         // com.rail.Ticket
    com.payments.Ticket pay = new com.payments.Ticket(); // FQCN
    System.out.println(rail.info() + " | " + pay.info());
  }
}
```

Putting it together: arrays and loops across packages

- Let’s look at an example with one pattern and three domains
    - TrainTicket[] (rail), Payable[] (payments), Notifier[] (notify)
- Boundary
    - Guard null entries

```
import com.rail.SleeperTicket;
import com.rail.AcTicket;
import com.payments.Payable;
import com.payments.CreditCard;
import com.payments.Upi;
import com.notify.Notifier;
import com.notify.SmsNotifier;

class App {
  public static void main(String[] args) {
    TrainTicket[] ts = { new SleeperTicket(), new AcTicket() };
    for (TrainTicket t : ts) System.out.println("fare -> " + t.fare(8));

    Payable[] ps = { new CreditCard(), new Upi(), null };
    for (Payable p : ps) { if (p == null) continue; System.out.println("paid ->
" + p.pay(300)); }

    Notifier[] ns = { new SmsNotifier() };
    for (Notifier n : ns) n.send("Delivered");
  }
}
```

Abstraction vs interface: pick the right tool

- Abstract class
  - Use when implementations share state or common code
  - Enforce required methods via abstract signatures
- Interface
  - Use for roles/capabilities across unrelated classes
  - Add small helpers with default, utilities with static
- Rules to follow
  - New field is private and validated in the child constructor
  - Override exactly one method; call super(...) and act only on success
- Acceptance checks
  - CashbackWallet 1% adds ₹1 on a ₹100 successful addMoney
  - PrepaidMetroPass blocks the 4th ride when the limit is 3

```
abstract class TrainTicket {
  abstract int fare(int km);
  String coachType(){ return "Sleeper"; }
}

interface Notifier { void send(String msg); }
interface Payable  { int  pay(int amount); }
```

That’s for today!
Any questions?