GUVI | HCL
Skill Up. Level Up

Intro to OOP: Classes & Objects and More!

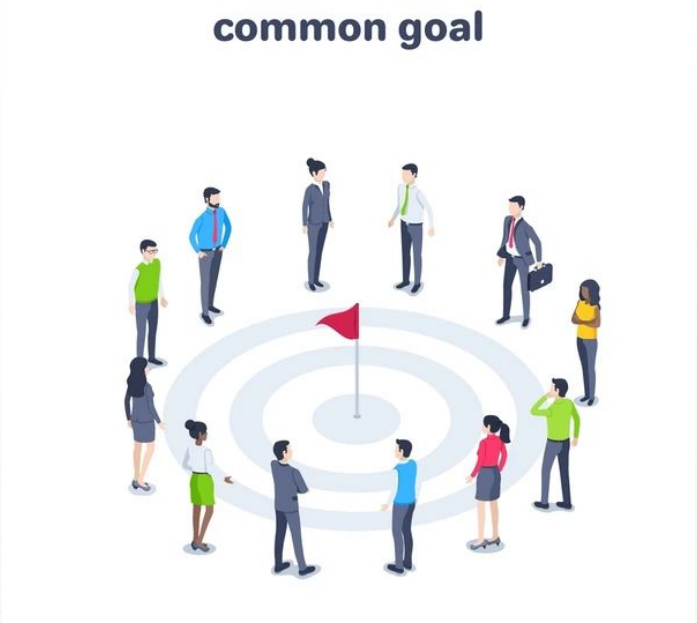
Programming Foundation

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

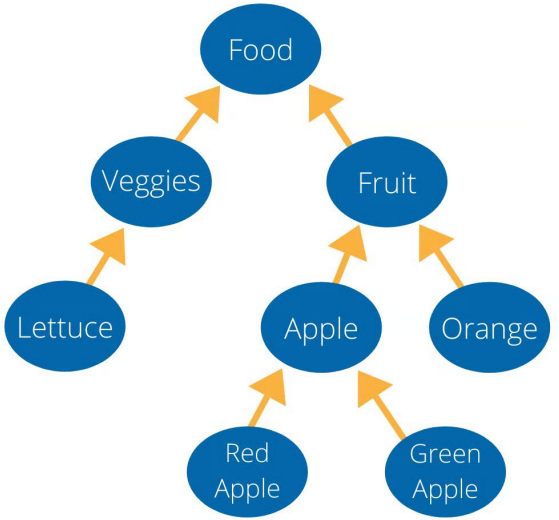
- What you will learn today
    - An introduction to Object Oriented Programming
    - Understanding Classes and Objects
    - Encapsulation
    - Inheritance
- How this builds on Lecture 3 - 5
    - Classes will reuse the concept of methods
    - Variables & data types will exist slightly differently in classes



OOP

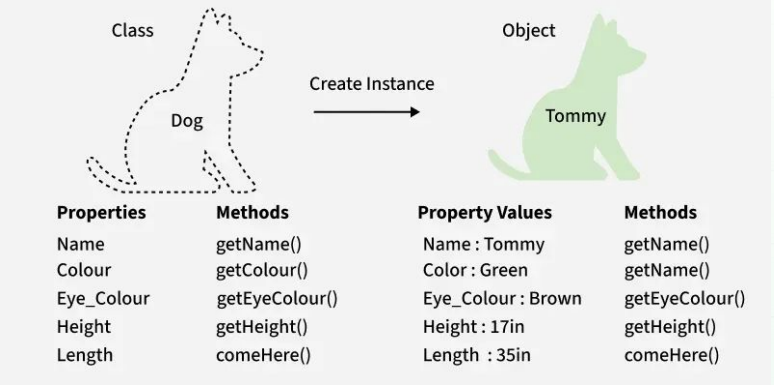
Why OOP?

- Real programs model “things” with data and actions
    - MetroPass, UPI wallet, Tiffin plan feel natural to reason about
- Procedural code spreads rules across places
    - Hard to change, easy to break hidden assumptions
- Object-Oriented Programming groups state and behavior
    - Blueprint first, then many usable copies
- We gain guardrails via clear names and small methods
    - Fewer surprises, easier debugging



What is a Class? What is an Object?

- Class
  - A blueprint listing fields for data and methods for actions
- Object
  - A concrete thing built from a class with its own values
- Fields
  - Named data the object carries, like balance or ownerName
- Methods
  - Named actions the object can do, like recharge or deductFare
- Independence rule
  - Two objects of the same class hold different field values
- Order of work
  - Define the class first, then create objects using the class



Anatomy of a Class

- Class name
    - Descriptive, PascalCase, matches the filename
- Fields
    - Data as nouns; start simple; set safe defaults where possible
- Methods
    - Verbs that read or change fields; one clear job per method
- Constructor
    - Special method that sets up a new object at creation time
- Access modifier
    - Keyword that controls who can see a member - public, private, & protected
- Recommended order inside a class
    - Fields, then constructors, then methods

Example: MetroPass Blueprint

- Purpose
  - A compact class with two fields and two actions to anchor ideas
- Fields (aka instance variables)
  - ownerName as text, balance as whole number with non negative rule
- Methods
  - addMoney increases balance only for positive amount
  - rideOnce deducts fare only if balance is enough
- Ordering inside the class
  - Fields, then constructors, then methods
- Boundary rules
  - Null ownerName becomes empty string
  - Zero or negative amounts are rejected

```
class MetroPass {
  private String ownerName;
  private int balance;

  MetroPass(String ownerName, int balance) {
    this.ownerName = (ownerName == null) ? "" :
ownerName.trim();
    if (balance < 0) balance = 0;
    this.balance = balance;
  }

  boolean addMoney(int amount) {
    if (amount <= 0) return false;
    balance += amount;
    return true;
  }

  boolean rideOnce(int fare) {
    if (fare <= 0 || fare > balance) return false;
    balance -= fare;
    return true;
  }
}
```

Creating Objects and Using Them

- Object creation
  - Use new with a constructor to build an instance
- Independence rule
  - Each object tracks its own balance and ownerName
- Call style
  - Use dot to call actions on a specific object
- Order of work
  - Declare variable, create object, then call methods
- Edge checks
  - Exact fare drops balance to zero
  - Zero or negative amount returns false and makes no change

```
MetroPass m1 = new MetroPass("Asha", 100);
MetroPass m2 = new MetroPass("Ravi", 50);

m1.addMoney(40);       // m1 balance becomes 140
m2.rideOnce(30);       // m2 balance becomes 20
m1.rideOnce(140);      // true, m1 balance becomes 0
boolean ok = m2.addMoney(0); // false, no change
```

Micro-activity: Spot Fields vs Methods

- Task
  - Classify each line as field or method
- Lines to classify
  - balance
  - ownerName
  - addMoney(amount)
  - rideOnce(fare)
  - dailyRideLimit
- Rules to recall
  - Fields are nouns and hold data
  - Methods are verbs and do actions

Constructors: Purpose and Defaults

- Constructor
  - Special method with class name that sets up a new object
- Default constructor
  - No inputs and sets safe starting values for fields
- Why defaults
  - Avoid half built objects and surprise nulls
- Boundary rules
  - Strings trimmed later and null becomes empty string
  - Numbers below zero become zero

```
class MetroPass {
  private String ownerName;
  private int balance;

  MetroPass() {
    this.ownerName = "";
    this.balance = 0;
  }
}
```

Parameterized Constructors

- What's a parameterized constructor?
    - Inputs set fields at creation for meaningful objects
- Guard checks
    - Correct bad inputs before storing to protect invariants
- Order of work
    - Validate inputs then assign to fields in one place
- Boundary rules
    - perKmFare below one becomes one
    - startKm below zero becomes zero

```
class AutorickshawMeter {
  private int startKm;
  private int perKmFare;

  AutorickshawMeter(int startKm, int perKmFare) {
    this.startKm = (startKm < 0) ? 0 : startKm;
    this.perKmFare = (perKmFare < 1) ? 1 :
perKmFare;
  }
}
```

The *this* Keyword

- this
  - Reference to the current object used inside its methods
- When you need it
  - Parameter names match field names
  - Syntax: this.field = param;
- Disambiguation rule
  - Use this.field when parameter names match field names
- Safety
  - Prevents shadowing that leaves fields unchanged
- Common bug
  - planName = planName leaves the field unchanged due to shadowing

```
class TiffinSubscription {
  private String planName;
  private int remainingMeals;

  TiffinSubscription(String planName, int
remainingMeals) {
    this.planName = (planName == null) ? "" :
planName;
    this.remainingMeals = Math.max(0,
remainingMeals);
  }
}
```

Encapsulation

Encapsulation: From Safe Starts to Safe Updates

- Problem
  - Public fields lets outside code break your rules after construction of the object
- Encapsulation
  - Keep fields private and expose narrow methods for access
- Read path
  - Getters return state without letting callers edit it directly
- Write path
  - Setters and mutators change state only when inputs pass checks
- Behavior contract
  - Mutators return boolean for success and do not print inside logic

```
class UPIWallet {
  private int balance;

  boolean addMoney(int amount) {
    if (amount <= 0) return false;
    balance += amount;
    return true;
  }

  boolean deduct(int amount) {
    if (amount <= 0 || amount > balance)
return false;
    balance -= amount;
    return true;
  }

  int getBalance() { return balance; }
}
```

Getters and Setters with Validation Rules

- Getter
  - Read-only window to a private field
- Setter or mutator
  - Changes state only after checks pass and returns boolean
- String rules
  - Trim spaces and convert null to empty string
- Number rules
  - Reject negatives and clamp when needed
- No I/O in logic
  - Return a value the caller can test

```
class TiffinSubscription {
  private String planName = "";
  private int remainingMeals = 0;

  String getPlanName() { return planName; }
  int getRemainingMeals() { return remainingMeals; }

  boolean setPlanName(String name) {
    String cleaned = (name == null) ? "" : name.trim();
    if (cleaned.isEmpty()) return false;
    planName = cleaned;
    return true;
  }
}
```

Example: UPIWallet with changePin

- Goal
  - Protect a sensitive field with a rule based mutator
- Fields
  - balance as private number
  - pin as private text
- changePin rules
  - oldPin must match current pin
  - newPin trimmed, length at least four, not equal to oldPin
- addMoney rule
  - Positive amounts only and returns boolean

```
class UPIWallet {
  private int balance = 0;
  private String pin = "1234";

  int getBalance() { return balance; }

  boolean addMoney(int amount) {
    if (amount <= 0) return false;
    balance += amount;
    return true;
  }

  boolean changePin(String oldPin, String newPin) {
    String n = (newPin == null) ? "" : newPin.trim();
    if (oldPin == null || !oldPin.equals(pin)) return false;
    if (n.length() < 4 || n.equals(pin)) return false;
    pin = n;
    return true;
  }
}
```

Activity: Model and Encapsulate a FASTagAccount

- Goal
    - Build a small class with private fields and safe updates
- Design and rules
    - Fields
        - vehicleNumber as text, balance as whole number not below zero
    - Constructor
        - Parameterized or default?
    - Getters
        - getVehicleNumber and getBalance for read only access
    - Setters and actions
        - setVehicleNumber trims text and rejects blank
        - addMoney(amount) only if amount is positive
        - payToll(fee) only if fee is positive and balance is enough

Inheritance

Inheritance: Reuse with extends

- Inheritance
  - A class reuses another class's code and adds or tweaks behavior
- Parent and child
  - Parent holds shared fields and methods
  - Child gains those and can add its own
- extends keyword
  - Declares that one class derives from another
- When it helps
  - Two classes share most behavior and only a small rule differs

```
class PrepaidMetroPass extends MetroPass {
  private int dailyRideLimit;
  private int ridesToday;

  boolean rideOnceWithLimit(int fare) {
    if (ridesToday >= dailyRideLimit) return false;
    boolean ok = rideOnce(fare); // reuse parent rule
    if (ok) ridesToday++;
    return ok;
  }
}
```

super: Parent Constructor and Method Calls

- super in a constructor
  - Calls the parent's constructor to set up shared fields
- super in a method
  - Calls a parent method and then adds extra steps in the child
- Order matters
  - Initialize parent first, then set child fields
- Example
  - PrepaidMetroPass passes owner and balance to MetroPass using super

```
class PrepaidMetroPass extends MetroPass {
  private int dailyRideLimit;
  private int ridesToday;

  PrepaidMetroPass(String ownerName, int balance, int dailyRideLimit) {
    super(ownerName, balance);            // parent setup
    this.dailyRideLimit = Math.max(0, dailyRideLimit);
    this.ridesToday = 0;
  }

  @Override
  boolean rideOnce(int fare) {            // override parent rule safely
    if (ridesToday >= dailyRideLimit) return false;
    boolean ok = super.rideOnce(fare);    // reuse parent check and deduct
    if (ok) ridesToday++;
    return ok;
  }
}
```

Overriding: adjust behavior in a child class

- What is it?
  - A child supplies its own version of a method defined in the parent
- Signature rule
  - Same name, parameters, and return type as the parent method
- Use @Override annotation
  - Ensures the method correctly overrides a parent method, catching errors (eg: incorrect method signature) at compile time
- Reuse, then extend
  - Call super.method(...) to keep parent checks, then add the new rule
- Keep the contract
  - Do not weaken input rules or change success meaning

```
// Child keeps parent checks, then adds 1% cashback
class CashbackWallet extends UPIWallet {
  @Override
  boolean addMoney(int amount) {
    if (!super.addMoney(amount)) return false;  // parent guards
    int bonus = amount / 100;                   // integer rupees
    if (bonus > 0) super.addMoney(bonus);       // reuse parent rule
    return true;
  }
}
```

Activity: specialize a class (inheritance)

- Goal
  - Create a child that adds one field and one rule while reusing parent logic
- Pick one base
  - UPIWallet: CashbackWallet with cashbackRate
  - TiffinSubscription: BonusTiffin with weeklyBonus
  - MetroPass: PrepaidMetroPass with dailyRideLimit
- 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

```
class PrepaidMetroPass extends MetroPass {
  private int dailyRideLimit;
  private int ridesToday;

  PrepaidMetroPass(String owner, int balance, int
limit) {
    super(owner, balance);
    this.dailyRideLimit = Math.max(0, limit);
    this.ridesToday = 0;
  }

  @Override
  boolean rideOnce(int fare) {
    if (ridesToday >= dailyRideLimit) return false;
    boolean ok = super.rideOnce(fare);
    if (ok) ridesToday++;
    return ok;
  }
}
```

Wrap-up and what’s next

- Today’s kit
  - Class as blueprint, object as instance
  - Constructors and this for clean setup
  - Encapsulation with getters and setters
  - Inheritance, super, and safe overriding
- Patterns to keep using
  - Nouns for fields and verbs for methods
  - Guard first, then update state
- Coming up
  - More OOP features!

That’s for today!
Any questions?