Advanced Programming

OOP Concepts: Classes, Encapsulation, Constructors, Scope, Composition, Inheritance, Polymorphism, Debugging

Java Python JavaScript

Classes & Objects

Java Implementation

Class

A blueprint defining the structure (attributes) and behaviors (methods) of a potential object.

Object

An instance of a class created at runtime using the `new` keyword. It holds specific state.

"Objects are nouns, methods are verbs."

Person.java

```java
public class Person {
    // Instance variables (attributes)
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method (behavior)
    public void introduce() {
        System.out.println("Hi, I'm " + name + ", age " + age);
    }
}

// Usage
Person alice = new Person("Alice", 30);
alice.introduce();
```

Classes & Objects

Python Implementation

Class Structure

Python classes use `__init__` to initialize objects. There is no strict type declaration for attributes.

The 'self' Reference

Methods explicitly take `self` as the first parameter, representing the instance calling the method.

"Explicit is better than implicit." - The Zen of Python

person.py

```python
class Person:
    # Constructor
    def __init__(self, name, age):
        # Instance variables initialized on self
        self.name = name
        self.age = age

    # Method (behavior)
    def introduce(self):
        print(f"Hi, I'm {self.name}, age {self.age}")

# Object creation (no 'new' keyword)
alice = Person("Alice", 30)

# Method call
alice.introduce()
```

Classes & Objects

JavaScript (ES6+) & React

ES6 Classes

JavaScript classes are syntactic sugar over prototype-based inheritance. They provide a cleaner, more familiar syntax for creating objects and dealing with inheritance.

Components

In React, class components extend `React.Component`. While function components are now preferred, understanding classes is crucial for legacy code.

"Syntactic sugar causes cancer of the semicolon." — Alan Perlis

Person.js

```
class Person {
    constructor(name, age) { // Constructor
        this.name = name;
        this.age = age;
    }

    introduce() { // Method
        console.log(`Hi, I'm ${this.name}, age ${this.age}`);
    }
}

// Usage
const alice = new Person("Alice", 30);
alice.introduce();

// React Component (Historical Context)
class User extends React.Component { ... }
```

Encapsulation

Getters & Setters (Java)

Private State

Data hiding is achieved by declaring class variables as `private`. This prevents direct unauthorized access from outside the class.

Public Accessors

Expose controlled access via **Getters** (read) and **Setters** (write). Setters allow you to add validation logic to protect data integrity.

BankAccount.java

```java
public class BankAccount {
    // Encapsulated (private) field
    private double balance;

    public BankAccount(double initialBalance) {
        if (initialBalance ⩾ 0) {
            this.balance = initialBalance;
        }
    }

    // Getter method
    public double getBalance() {
        return balance;
    }

    // Setter with validation
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
}
```

Encapsulation

Python Properties

Convention

Python uses a single underscore prefix (e.g., `_balance`) to indicate "protected" or internal variables.

@ Properties

Instead of explicit getters/setters, use `@property`. This allows method-like logic (validation) with attribute-like access syntax.

"We are all consenting adults here." — Python's philosophy on access control.

bank_account.py

```python
class BankAccount:
    def __init__(self, initial=0):
        self._balance = initial if initial >= 0 else 0

    @property
    def balance(self): # Getter
        return self._balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount

    def withdraw(self, amount):
        if amount > 0 and amount <= self._balance:
            self._balance -= amount
            return True
        return False
```

Encapsulation

JavaScript Private Fields

Private Fields (#)

Modern JavaScript (ES2022) introduces true privacy using the `#` prefix. These fields cannot be accessed from outside the class.

Safety & Validation

By hiding internal state, we force usage of public methods (getters/setters), ensuring rules (like positive deposits) are always enforced.

"Before #private, we relied on closures or the '_underscore' convention."

BankAccount.js

```
class BankAccount {
    #balance = 0;    // Private field (Modern JS)

    constructor(initialBalance) {
        if (initialBalance ⩾ 0) {
            this.#balance = initialBalance;
        }
    }

    getBalance() {  // Getter
        return this.#balance;
    }

    deposit(amount) {
        if (amount > 0) {
            this.#balance += amount;
        }
    }
}

// Usage
const account = new BankAccount(100);
// account.#balance // Syntax Error: Private field
console.log(account.getBalance()); // 100
```

Constructors

Java Overloading & Chaining

Overloading

Defining multiple constructors with different parameter lists. This provides flexibility in how objects are initialized (e.g., empty vs. specific values).

Constructor Chaining

Using `this(...)` to call another constructor within the same class. This reduces code duplication by centralizing initialization logic.

"Don't repeat yourself (DRY) applies to constructors too."

Rectangle.java

```java
public class Rectangle {
    private double width, height;

    // 1. Default constructor (Chaining)
    public Rectangle() {
        this(1.0, 1.0); // Calls parameterized
    }

    // 2. Parameterized constructor (The Logic)
    public Rectangle(double width, double height) {
        this.width = Math.max(0, width);
        this.height = Math.max(0, height);
    }

    // 3. Copy constructor
    public Rectangle(Rectangle other) {
        this(other.width, other.height);
    }
}
```

Constructors

Python Alternatives

Default Arguments

Instead of multiple constructors, Python uses `__init__` with default parameter values to handle optional initialization data.

Factory Methods

Use `@classmethod` to define alternative "constructors" that return an instance of the class (e.g., creating a Square from a Rectangle class).

"Explicit is better than implicit." — The Zen of Python

Rectangle.py

```python
class Rectangle:
    # 1. Flexible Init with defaults
    def __init__(self, width=1.0, height=1.0):
        self.width = max(0, width)
        self.height = max(0, height)

    # 2. Alternative Constructor (Factory)
    @classmethod
    def square(cls, side):
        return cls(side, side)

    @classmethod
    def from_dict(cls, data):
        return cls(data['width'], data['height'])

# Usage
rect1 = Rectangle(3, 4)
sq_obj = Rectangle.square(5) # Creates 5x5 rectangle
```

Parameter Passing

Language Comparison

Java

STRATEGY

Pass-by-Value

(References are copied)

- **Primitives:** Copied directly. Changes inside method don't affect original.
- **Objects:** Reference is copied. Can modify object state, but cannot change the original reference.

```
void swap(Object a, Object b) {
  Object temp = a; a = b; b = temp;
} // No effect outside
```

Python

STRATEGY

Pass-by-Object-Ref

(Everything is a reference)

- **Mutability Matters:** Mutable objects (lists, dicts) can be changed.
- **Immutable:** Ints, strings, tuples cannot be changed in place (new object created).

```
def modify(lst):
  lst.append(1) # Changes original
  lst = [] # Local rebind only
```

JavaScript

STRATEGY

Pass-by-Value

(Similar to Java)

- **Primitives:** Passed by value (number, string, boolean).
- **Objects:** Reference value is passed. Can mutate properties, cannot reassign reference.

```
function update(obj) {
  obj.x = 5; // Modifies original
  obj = {x: 5}; // No effect outside
```

Scope & Visibility

Java Access Modifiers

`private`

Visible **only** within the class itself. Most restrictive.

`(default)`

Package-private. Visible to classes in the **same package** only.

`protected`

Visible to same package **plus subclasses** (even in different packages).

`public`

Visible **everywhere**. Least restrictive.

Employee.java

```java
public class Employee {
    private String ssn; // Class-internal only

    String department; // Package-private

    protected String name; // Subclasses + pkg

    public String id; // Everywhere

    private void validateSSN() { }
}
```

Best Practice: Use the most restrictive access level that makes sense (Principle of Least Privilege). Start with `private` and open up as needed.

Scope & Visibility

Python Conventions & Name Mangling

Public (Default)

No underscores. Variables and methods are accessible from **anywhere**. This is the Pythonic standard.

_protected

Single underscore prefix. A strong convention indicating "internal use only." No runtime enforcement prevents access.

__private

Double underscore prefix. Triggers *name mangling* (`_Class__var`) to prevent accidental overrides in subclasses. Harder, but not impossible, to access.

employee.py

```python
class Employee:
    def __init__(self, ssn, name):
        self._ssn = ssn # Protected (Convention)
        self.__name = name # Private (Mangled)
        self.role = "Dev" # Public

    def _internal_method(self): # Protected
        pass

    def __private_method(self): # Private
        pass
```

Python Philosophy: *"We are all consenting adults." Privacy relies on adhering to conventions rather than strict compiler enforcement.*

Scope & Visibility

JavaScript Patterns & Conventions

_underscore

Convention only. Properties prefixed with _ signal "internal use". Not truly private, but respected by developers.

Closures

True privacy via scope. Variables defined inside the constructor (using `let/const`) are inaccessible from outside.

#privateFields

Modern Standard (ES2022). Native syntax using # prefix. Strictly enforces encapsulation at runtime.

Employee.js

```
class Employee {
    constructor(ssn, name) {

        // 1. Private via closure (pre-ES2022)
        let _ssn = ssn;

        // 2. Conventionally protected
        this._name = name;
        this.publicId = 'EMP123';

        // Method can access closure variable
        this.getSSN = () => _ssn;
    }
}
```

*Note: While **closures** were the standard for years, modern JS codebases increasingly adopt **#private** fields for cleaner class definitions and better performance.*

Association & Containment

Java Implementation (Composition)

Composition (HAS-A)

A strong relationship where one object contains another. The contained object is an integral part of the container.

Code Reuse

Instead of inheriting functionality, classes are built by combining simpler objects. An `Employee` **has an** `Address`.

"Favor composition over inheritance for flexibility."

Employee.java

```java
class Address {
    private String street, city;
    public Address(String street, String city) {
        this.street = street;
        this.city = city;
    }
}

public class Employee {
    private String name;
    private Address address; // Containment (HAS-A)

    public Employee(String name, Address address) {
        this.name = name;
        this.address = address; // Employee owns Address
    }

    public void showInfo() {
        System.out.println(name + " lives at " + address);
    }
}
```

Association & Containment

Python Composition Pattern

Composition

A "HAS-A" relationship where complex objects are built from smaller objects. For example, an Employee *has an* Address.

Code Reuse

Python relies heavily on composition over inheritance to create flexible, maintainable code structures that are easier to test.

"Favor object composition over class inheritance."

composition.py

```python
class Address:
    def __init__(self, street, city):
        self.street = street
        self.city = city

class Employee:
    def __init__(self, name, address):
        self.name = name
        self.address = address # Composition (Has-A)

    def get_info(self):
        return f"{self.name} lives in {self.address.city}"

# Usage
home = Address("123 Tech Lane", "PyCity")
emp = Employee("Dev Dave", home)

# Accessing composed object
print(emp.get_info()) # Output: Dev Dave lives in PyCity
```

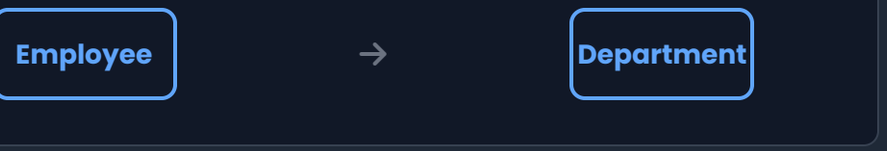
Association vs. Containment

Understanding Object Relationships

Association

"KNOWS ABOUT"

- **Weak Coupling:** Objects have independent lifecycles. If the main object is destroyed, the associated object survives.
- **Reference Only:** Usually passed via constructor or method arguments.




```
// Employee refers to existing Department
class Employee {
  private Department dept;

  // Passed in, not created here
  public Employee(Department d) {
    this.dept = d;
  }
}
```

Containment

"HAS A" (OWNERSHIP)

- **Strong Ownership:** The container controls the lifecycle. Part cannot exist without the Whole.
- **Exclusive:** The part belongs to one container only.


```
// Car creates and owns Engine
class Car {
  private Engine engine;

  public Car() {
    // Created internally
    this.engine = new Engine();
  }
}
```

Code Reuse via Composition

Prefer "HAS-A" over "IS-A"

Composite Pattern

Flexible Relationships

Unlike inheritance (static), composition allows you to change behavior at runtime by swapping components.

Loose Coupling

Classes don't depend on implementation details of parent classes. They just use the public interface of the composed object.

Testability

Easier to unit test. You can inject mock objects (like a fake Logger) into the constructor.

OrderService.java

```java
class Logger {
    public void log(String msg) {
        System.out.println("[LOG] " + msg);
    }
}

class OrderService {
    private Logger logger; // HAS-A Logger

    // Dependency Injection via Constructor
    public OrderService(Logger l) {
        this.logger = l;
    }

    public void processOrder(Order o) {
        logger.log("Processing " + o.getId());
    }
}
```

*Design Principle: **"Favor composition over inheritance."** It prevents the "fragile base class" problem and keeps classes focused on a single responsibility.*

Code Reuse via Mixins

Python Implementation

Mixin Pattern

A class designed to provide specific capabilities to other classes via inheritance. Mixins are not intended to stand alone or be instantiated directly.

Composition

Python's multiple inheritance allows "mixing in" functionalities. This promotes flat class hierarchies and code reuse across unrelated classes.

"Favor composition over inheritance (except for Mixins)."

mixins.py

```python
class LoggerMixin:
    def log(self, message):
        print(f"[LOG] {message}")

# Multiple inheritance (Mixin provides functionality)
class OrderService(LoggerMixin):
    def process_order(self, order):
        # 'log' method is available from Mixin
        self.log(f"Processing {order.id}")
        # ... business logic ...

# Usage
service = OrderService()
# Example order object stub
service.process_order(order_obj)
```

Debugging OOP Code

Common Issues & Solutions

Java

MAJOR PITFALL

NullPointerError

(Accessing null references)

Solution: Explicit null checks before method calls or property access.

Scope: Strict access modifiers (private, protected) prevent accidental external modification.

```
if (obj ≠ null) {
  obj.doSomething();
} // Safe access
```

Python

MAJOR PITFALL

AttributeError

('NoneType' object has no attribute)

Solution: Check if object is not None before usage.

Naming: Watch for `__private` name mangling which makes attributes harder to access.

```
if obj is not None:
  obj.do_something()
# Handles NoneType safely
```

JavaScript

MAJOR PITFALL

Wrong 'this' Binding

(Context loss in callbacks)

Solution: Use Arrow Functions `() =>` to lexically bind `this`.

Scope: Use `let/const` to create block scopes and avoid leakage.

```
// Arrow preserves 'this'
const action = () ⇒ {
  this.method();
}
```

Debugging Techniques

Strategies & Best Practices

Structured Logging

Use levels (`DEBUG`, `INFO`, `ERROR`) instead of print statements to control noise.

Assertions

Validate **invariants** and assumptions. Fail fast during development if data state is invalid.

Breakpoints

Pause execution to inspect variable states and call stacks in real-time rather than guessing.

Small Methods

Isolate logic into small, testable units. Easier to pinpoint where logic breaks.

DebuggingExample.java

```java
public void processTransaction(double amount) {

    // 1. Logging entry
    logger.debug("Start transaction: " + amount);


    // 2. Assertion (Development check)
    assert amount > 0 : "Amount must be positive!";


    this.balance += amount;


    // 3. Invariant check
    assert this.balance ⩾ 0 : "Negative balance!";
}
```

Note: *In Java, assertions must be explicitly enabled with the `-ea` flag. In Python, they can be optimized out with `-O`.*

Inheritance

Java Implementation (extends)

The 'extends' Keyword

Establishes an "IS-A" relationship. The child class (Subclass) acquires all non-private fields and methods of the parent class (Superclass).

Overriding & Super

`@Override` redefines inherited behavior. `super` accesses the parent's members or constructors.

"Java supports single class inheritance but multiple interface implementation."

InheritanceExample.java

```java
class Animal {
    String name;
    Animal(String n) { name = n; }
    void eat() { System.out.println(name + " is eating"); }
}

// Child class inherits from Animal
class Dog extends Animal {
    Dog(String n) {
        super(n); // Call parent constructor
    }

    @Override
    void eat() {
        super.eat(); // Reuse parent logic
        System.out.println("...with dog food");
    }

    void bark() {
        System.out.println(name + " says Woof!");
    }
}
```

Inheritance

Python Implementation

Parent & Child

Classes inherit attributes and methods by passing the parent class name in parentheses: `class Child(Parent):` .

super() Function

The `super()` function provides access to inherited methods, essential for initializing parent classes in `__init__` .

"Python uses Method Resolution Order (MRO) to handle inheritance."

inheritance_demo.py

```python
class Animal:
    def __init__(self, name):
        self.name = name
    def eat(self):
        print(f"{self.name} is eating")

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name) # Call parent constructor

    def eat(self): # Method override
        super().eat()
        print("...with dog food")

    def bark(self):
        print(f"{self.name} says Woof!")

# Usage
my_dog = Dog("Buddy")
my_dog.eat(); my_dog.bark()
```

Inheritance

JavaScript (ES6) Implementation

extends Keyword

Syntactic sugar over prototype-based inheritance. It allows a class to inherit methods and properties from another class, establishing an "is-a" relationship.

↑ super() Call

Essential in derived classes. You must call `super()` before accessing `this`. Use `super.method()` to access parent methods.

inheritance.js

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    eat() {
        console.log(`${this.name} is eating`);
    }
}

class Dog extends Animal {
    constructor(name) {
        super(name); // Call parent constructor
    }
    eat() {
        super.eat(); // Call parent method
        console.log("...with dog food");
    }
    bark() {
        console.log(`${this.name} says Woof!`);
    }
}

// Usage
const myDog = new Dog("Buddy");
myDog.eat();
```

Inheritance Hierarchy

Class Relationships & Structure

Overriding vs. Overloading

Java Polymorphism Mechanisms

Overriding

RUNTIME POLYMORPHISM

- **Same Signature:** Method name and parameters must match exactly.
- **Inheritance Required:** Happens between Parent and Child classes.


```java
// Child modifies behavior
class Dog extends Animal {
  @Override
  public void makeSound() {
    // Specific impl
    System.out.println("Woof!");
  }
}
```

Overloading

COMPILE-TIME POLYMORPHISM

- **Different Parameters:** Must differ in number, type, or order of arguments.
- **Same Class:** Usually occurs within a single class context.


```java
// Same name, different inputs
class Printer {
  void print(String text) { ... }

  // Overloaded
  void print(int number) { ... }
}
```

Super Keyword

Usage Across Languages

Java

REFERENCE

super

(Refers to superclass)

- **Constructor:** `super()` must be the first statement in the constructor body.
- **Methods:** Use dot notation to call overridden methods.

```
public Dog() {
  super(); // Parent ctor
}
super.eat(); // Parent method
```

Python

FUNCTION

super()

(Returns proxy object)

- **Constructor:** Call `__init__` explicitly on the proxy object.
- **Dynamic:** Resolves method resolution order (MRO) dynamically.

```
def __init__(self):
  super().__init__()

super().eat() # Parent method
```

JavaScript

KEYWORD

super

(Context dependent)

- **Constructor:** Must call `super()` before accessing this.
- **Methods:** Accesses parent class prototype methods.

```
constructor() {
  super(); // Before 'this'
}
super.eat(); // Parent method
```

Access Modifiers in Inheritance

Controlling Member Visibility in Subclasses

Java

STRICT ENFORCEMENT

public	INHERITED
protected	INHERITED
private	NOT INHERITED

```java
public class Parent {
  public int a; // Child can access
  protected int b; // Child can access
  private int c; // Hidden from Child
}

class Child extends Parent {
  void test() {
    this.c = 10; // Compile Error!
  }
}
```

Python

CONVENTION BASED

name	PUBLIC
_name	PROTECTED (CONVENTION)
__name	PRIVATE (MANGLED)

```python
class Parent:
  def __init__(self):
    self.a = 1 # Public
    self._b = 2 # Use internally only
    self.__c = 3 # Name mangled

class Child(Parent):
  def show(self):
    print(self._b) # Works (Convention)
    # print(self.__c) # AttributeError
    # Access via: self._Parent__c
```

Polymorphism in Action

Java Implementation

Upcasting

A `List<Animal>` can store instances of any subclass (`Dog`, `Cat`). The reference type is the parent, but the object in memory is the specific child.

Late Binding

The method implementation is chosen at **runtime** based on the actual object type, not the variable type. This is the core of polymorphism.

"Treat different objects uniformly through their common interface."

Main.java

```java
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Polymorphic List: Holds Animal references
        List<Animal> animals = List.of(
            new Dog("Buddy"),
            new Cat("Whiskers")
        );

        // Iterate and invoke polymorphic behavior
        for (Animal a : animals) {
            // Runtime decides: Dog.makeSound() or Cat.makeSound()
            a.makeSound();
        }
    }
}
// Output:
// Buddy says Woof!
// Whiskers says Meow!
```

Polymorphism in Action

Dynamic Typing vs Component Specialization

Python

DUCK TYPING

- **Implicit Interfaces:** "If it walks like a duck and quacks like a duck, it's a duck." No shared base class required.
- **Runtime Resolution:** Method existence is checked when called.

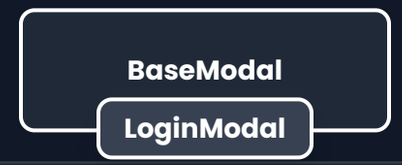

```python
# Objects behave same without inheritance
animals = [Dog(), Cat()]

for a in animals:
  # Works if method exists
  a.make_sound()
```

React

COMPONENT SPECIALIZATION

- **Shared Behavior:** Base components provide logic (state, lifecycle), while children provide specific UI.
- **Specialization:** A generic 'Modal' becomes a specific 'LoginModal'.




```jsx
// Child reuses logic, customizes render
class LoginModal extends BaseModal {
  render() {
    return (
      <div>Hi, {this.state.name}</div>
    );
  }
}
```

Inheritance vs. Composition

Choosing the Right Code Reuse Pattern

Inheritance

"IS-A" RELATIONSHIP

- **Rigid Hierarchy:** Subclasses are tightly coupled to the parent class implementation.
- **White-Box Reuse:** Parent internals are often visible to subclasses.


```
// Dog IS AN Animal
class Dog extends Animal {
  @Override
  void eat() {
    super.eat(); // Coupled
    print("kibble");
  }
}
```

Composition

"HAS-A" RELATIONSHIP

- **Flexible Reuse:** Behavior is delegated to components. Can be changed at runtime.
- **Black-Box Reuse:** Components interact only through public interfaces.


```
// Car HAS AN Engine
class Car {
  private Engine engine;

  void move() {
    engine.run(); // Delegate
  }
}
```

Complete Example

Library System (Java)

Abstract Base

`Item` defines the shared contract. It handles common state (title, availability) and forces subclasses to implement `checkOut()`.

Inheritance

`Book` extends `Item`, reusing the base logic while adding book-specific fields like `author`.

Polymorphism

The `@Override` annotation ensures `checkOut()` provides specific behavior for books.

LibrarySystem.java

```java
abstract class Item {
    protected String title;
    protected boolean available = true;
    public Item(String t) { title = t; }
    public abstract void checkOut();
}

class Book extends Item {
    private String author;
    public Book(String t, String a) {
        super(t);
        this.author = a;
    }
    @Override
    public void checkOut() {
        if (available) {
            available = false;
            System.out.println("Book '" + title + "' checked out");
        }
    }
}
```

Scalability: Adding a `DVD` class requires no changes to existing `Item` logic, just a new implementation of `checkOut()`.

bash — 80x24

```
$ compiling modules...
> Classes_and_Objects... OK
> Encapsulation... OK
> Inheritance... OK
> Polymorphism... OK
```

```
echo "Thank You!"
```

Thank You!

```
$ ready_for_questions
```

Questions?

Feel free to ask about any OOP concepts.

INSTRUCTOR

Nabajyoti Medhi