Java OOPS Reference Guide: The Employee Management System

This guide serves as a complete reference for the `OOPS_Demo_Project.java` file, explaining the four pillars of Object-Oriented Programming (OOP) in detail.

Project Analogy: The system models a company where the abstract **Employee** class provides common structure, and concrete classes (**Manager**, **Developer**) provide specific behaviors.

1. Abstraction (via `abstract class Employee`)

Definition: Abstraction focuses on defining *what* an object is supposed to do, hiding the complex *how*. It defines a contract that all subclasses must follow.

Code Element	OOP Concept	Detailed Explanation for Recall
`abstract class Employee`	Abstract Class	Cannot be instantiated directly. It serves as a blueprint, **forcing specific subclasses** to be created (e.g., you must be a `Developer`, not just a generic `Employee`).
`public abstract double calculateBonus();`	Abstract Method	A method declaration **without an implementation** (no body). This forces every non-abstract subclass (`Manager`, `Developer`) to provide its own unique logic for calculating the bonus.

Key Takeaway: Abstraction is about creating a hierarchy where essential functionality is **guaranteed**, but specialized implementation is deferred to concrete classes.

2. Encapsulation (Focus: `Employee` class data protection)

Definition: Encapsulation is the mechanism of bundling data (fields) and the methods (getters and setters) that operate on the data into a single unit (the class). This protects data integrity.

Code Element	OOP Concept	Detailed Explanation for Recall
`private int id;`	Private Fields	The data fields (`id`, `name`, `baseSalary`) are **hidden** from direct access outside the class. This is the core of data protection.
`public double getName()`	Getter (Accessor)	Provides controlled **read-only access** to the private data.
`public void setBaseSalary(...)`	Setter (Mutator)	Provides controlled **write access**. The method includes **validation** (`if (newSalary > 0)`), ensuring that the internal state (`baseSalary`) can only be modified with valid data, thereby protecting the object's integrity.

Key Takeaway: Encapsulation hides the implementation details and provides a public interface for interacting with the object's data, ensuring the object maintains a consistent and valid state.

3. Inheritance (Focus: `extends` keyword)

Definition: Inheritance is the process where one class (subclass/child) acquires the properties and methods of another class (superclass/parent). This promotes **code reusability**.

Code Element	OOP Concept	Detailed Explanation for Recall
`class Developer extends Employee`	Inheritance Link	The `Developer` class automatically gains all the non-private members of `Employee`. This is the **"is-a"** relationship (`Developer` *is an* `Employee`).
`super(id, name, baseSalary);`	Super Call	Used within the subclass constructor to explicitly call and initialize the parent class's constructor, setting up the inherited fields.
`public double getAnnualSalary()`	Reused Code	This method is defined only once in `Employee` but is available to both `Manager` and `Developer`, demonstrating code reuse.

Key Takeaway: Inheritance minimizes redundant code by allowing specialized classes to build upon the foundation of a general class.

4. Polymorphism (Many Forms)

Definition: Polymorphism is the ability for a single identifier (like a method name) to take on multiple forms or implementations.

4.1. Runtime Polymorphism (Method Overriding)

Code Element	OOP Concept	Detailed Explanation for Recall
`@Override public double calculateBonus()`	Method Overriding	The child classes (`Manager`, `Developer`) provide their own **specific implementation** for a method defined in the parent (`Employee`). The specific method chosen depends on the object type.
`List<Employee> allEmployees`	Parent Type Reference	A list of the parent type (`Employee`) holding instances of child types (`Manager`, `Developer`). This is the foundation of runtime polymorphism.
`emp.calculateBonus()` in loop	Dynamic Dispatch	When this line executes, the Java Virtual Machine (JVM) determines the actual type of `emp` (Manager or Developer) at runtime and calls the corresponding bonus method.

4.2. Compile-Time Polymorphism (Method Overloading)

Code Element	OOP Concept	Detailed Explanation for Recall

`displayInfo()` vs. `displayInfo(String role)`	Method Overloading	Defining multiple methods in the same class with the same name but **different parameter lists** (signatures). The compiler determines which method to call based on the arguments provided.