Advanced Programming (OOP)

Module 1: Programming Environment and OOP Introduction

SDB

IEM

Spring 2025

Module 1: Topics

- **Understanding the build system:** IDE, debugging, profiling, and source code management (hands-on in Lab).
- **Introduction to programming paradigms.**
- **Comparison of OOP with Procedural Paradigm.**
- **Advantages of Object-Oriented Programming.**

Outline

Introduction to Build System

What is a Build System?

- A set of tools and processes used to automate the compilation, linking, and packaging of source code into executable programs.
- Helps manage dependencies, streamline development, and ensure reproducibility.

Key Components of a Build System:

- **Integrated Development Environment (IDE):** Provides a user-friendly interface for writing, debugging, and testing code. *Examples: Eclipse, IntelliJ IDEA, NetBeans.*
- **Build Tools:** Automates tasks like compiling code, running tests, and generating executables. *Examples: Maven, Gradle.*
- **Version Control Systems:** Tools to track and manage changes in code over time. *Examples: Git, SVN.*

Debugging, Profiling, and Source Code Management

Debugging:

- Identifying and fixing errors in code using breakpoints, stack traces, and logs.
- Debuggers in IDEs help inspect variables, step through code, and locate issues.

Profiling:

- Analyzing the performance of a program to identify bottlenecks or inefficiencies.
- Tools like VisualVM or JProfiler provide insights into CPU usage, memory consumption, and execution time.

Source Code Management:

- Organizing and maintaining versions of source code across team.
- Key commands in Git: `git add`, `git commit`, `git push`, `git pull`.
- Platforms like GitHub and GitLab enhance collaboration.

Outline

Introduction to Programming Paradigms

What is a Programming Paradigm?

- A style or "way" of programming that defines how problems are solved.
- Paradigms influence the structure of code, approaches to problem-solving, and the design of software.
- Common paradigms include Procedural, Object-Oriented, Functional, and Event-Driven programming.

Introduction to Programming Paradigms

What is a Programming Paradigm?

- A style or "way" of programming that defines how problems are solved.
- Paradigms influence the structure of code, approaches to problem-solving, and the design of software.
- Common paradigms include Procedural, Object-Oriented, Functional, and Event-Driven programming.

Why Paradigms Matter:

- Paradigms determine the flow and design of your code.
- They guide how data is managed, how control flows through the program, and how code is structured for maintenance.
- The choice of paradigm affects the scalability, maintainability, and efficiency of the code.

Key Differences Between Paradigms

Procedural Programming:

- Focuses on functions or procedures that operate on data.
- Programs are typically written as sequences of steps.
- Example: C, Pascal.

Key Differences Between Paradigms

Procedural Programming:

- Focuses on functions or procedures that operate on data.
- Programs are typically written as sequences of steps.
- Example: C, Pascal.

Object-Oriented Programming:

- Centers around objects, which bundle both data and functions that operate on the data.
- Promotes modularity, reusability, and maintainability by organizing code into classes and objects.
- Example: Java, Python.

Procedural and OOP Paradigms: An Example I

Real-World Example: A Banking System
Procedural Approach:

```
double balance = 1000;

void deposit(double amount) {
    balance += amount;
}

void withdraw(double amount) {
    if (balance >= amount) {
        balance -= amount;
    } else {
        System.out.println("Insufficient
            Funds");
    }
}
```

Listing 1: Procedural Example

- Functions operate directly on global variables (e.g., `balance`).
- Limited modularity and difficult to extend (e.g., adding account types).

Procedural and OOP Paradigms: An Example II

Object-Oriented Approach:

```
class BankAccount {
    private double balance;

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

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            System.out.println("
                Insufficient Funds");
        }
    }
}
```

Listing 2: OOP Example

Why OOP Wins:

- Enhances code readability and maintainability.
- Protects data integrity and supports real-world modeling.
- Simplifies the addition of new features (e.g., interest calculation).

Key Differences Between Procedural and OOP I

	Procedural	**OOP**
Focus	Functions and procedures that operate on data.	Objects that encapsulate data and behavior.
Modularity	Code organized into functions.	Code organized into classes and objects.
Data Handling	Global data shared across functions.	Data encapsulated within objects (Encapsulation).
Code Reusability	Limited, achieved through function reuse.	High, through inheritance and polymorphism.
Scalability	Becomes complex for large systems.	Better suited for large, complex systems.
Real-World Modeling	Weak abstraction for real-world entities.	Strong abstraction through objects and classes.
Examples	C, Pascal.	Java, C++, Python.

Key Differences Between Procedural and OOP II

Procedural Programming:

- Focuses on functions and procedures.
- Data and functions are separate.
- Code reuse is limited to function calls.
- Better for simple, linear tasks.

Object-Oriented Programming:

- Focuses on objects that encapsulate data and behavior.
- Promotes code reusability through inheritance and polymorphism.
- Modular design makes it better for large and complex systems.
- Strongly aligns with real-world problem modeling.

Food for Thought: Paradigm Shift

- How does your thinking about a problem change when you switch from procedural to object-oriented programming?
- In what real-world situations might procedural programming be more beneficial than OOP?
- Are there tasks where mixing paradigms (hybrid programming) could lead to better solutions?

Outline

Introduction to Object-Oriented Programming

What is Object-Oriented Programming?

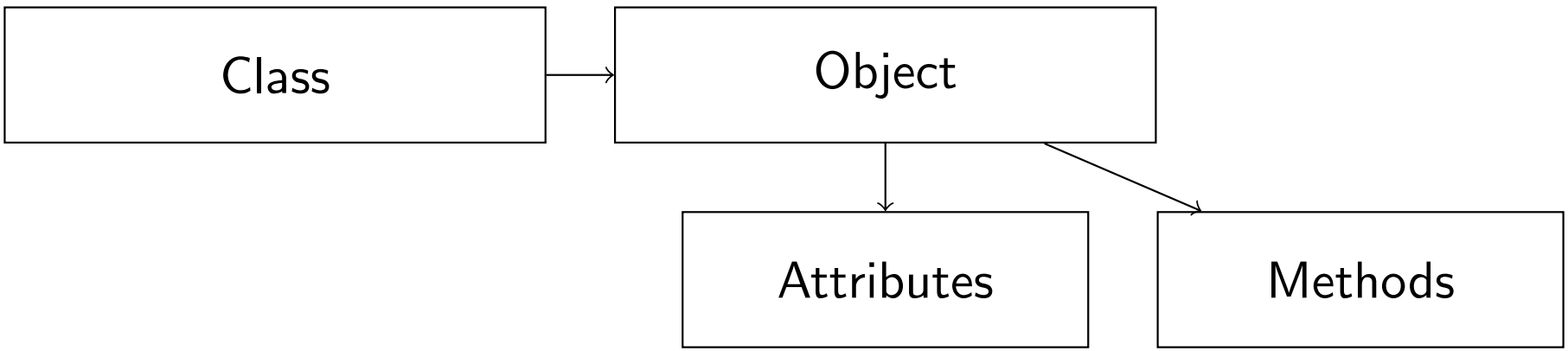
- A programming paradigm that revolves around the concept of objects and classes.
- Focuses on the interactions between objects and their properties.

What is a Class?

- A blueprint or template that defines the properties and behavior of an object.
- A class defines the structure and behavior of an object.

What is an Object?

- An instance of a class.
- Has its own set of attributes (data) and methods (functions).



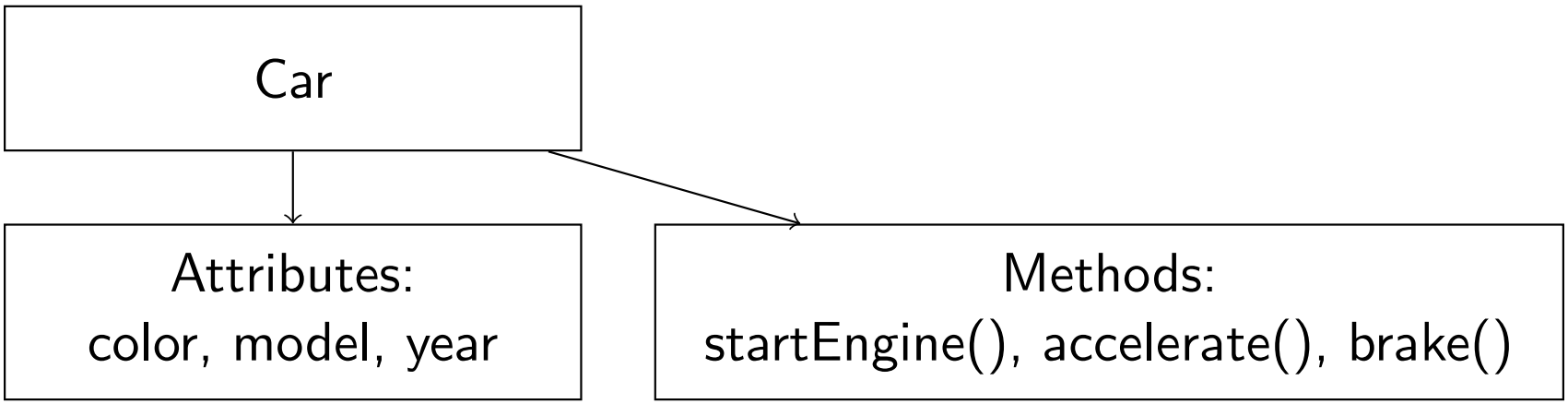
Need for Classes and Objects

Why do we need Classes and Objects?

- To model real-world entities and systems.
- To create reusable and modular code.
- To improve code maintainability and scalability.

Example:

- A car can be represented as an object with attributes like color, model, and year.
- A car can have methods like startEngine(), accelerate(), and brake().

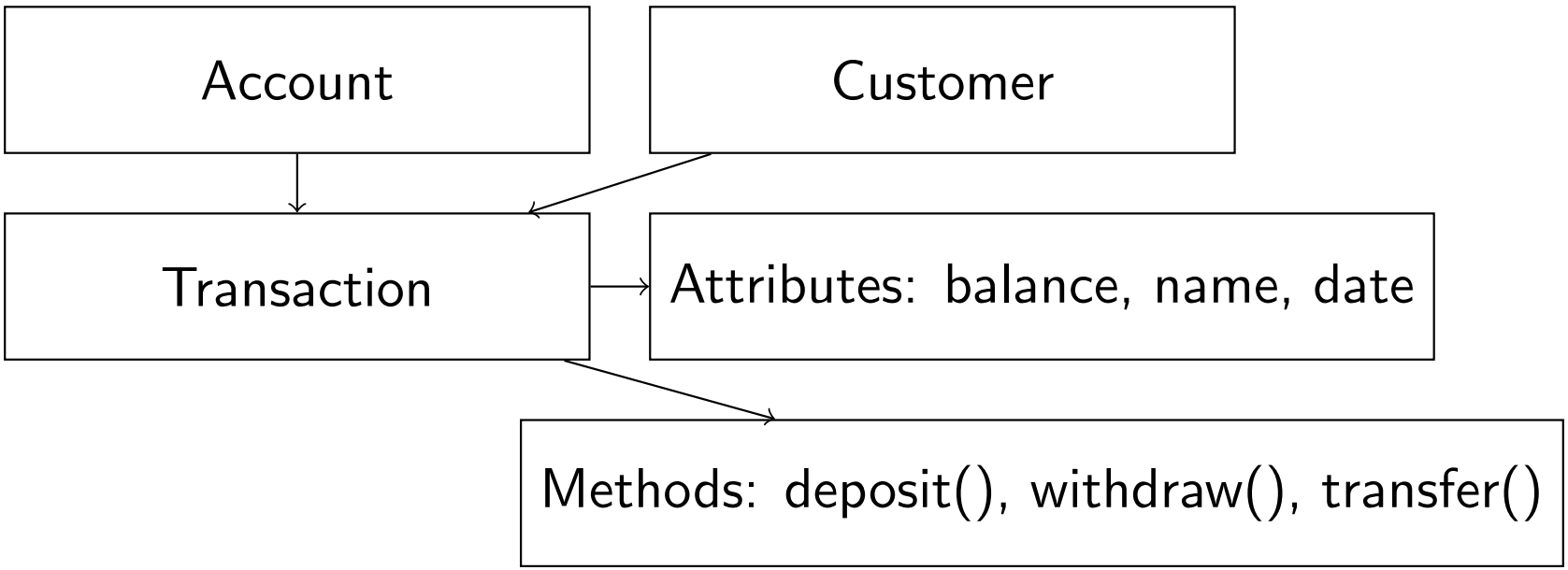


Benefits of Using Classes and Objects

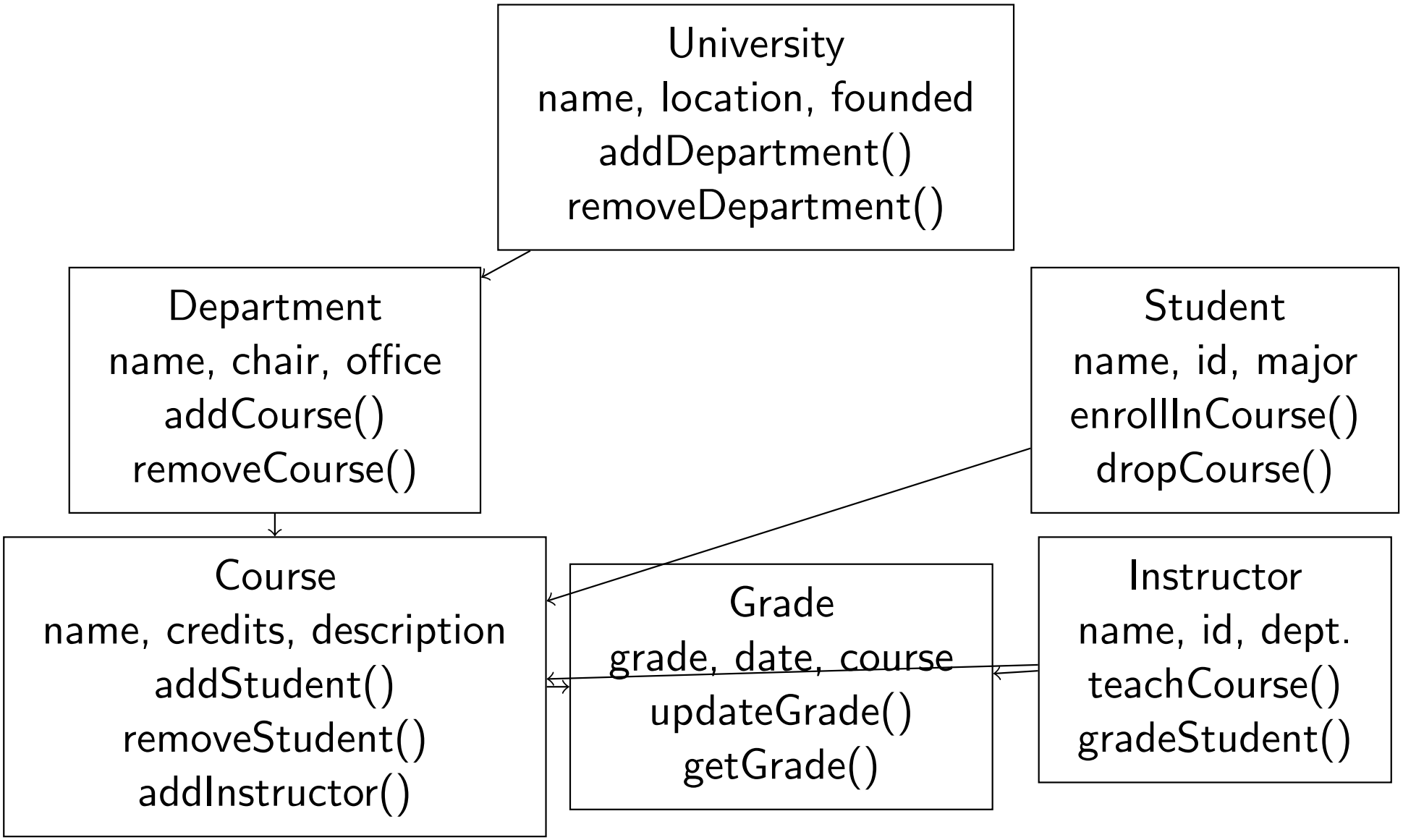
Benefits of Using Classes and Objects

- Improved code organization and structure.
- Easier to modify and maintain code.
- Reduced code duplication.
- Improved scalability and reusability.

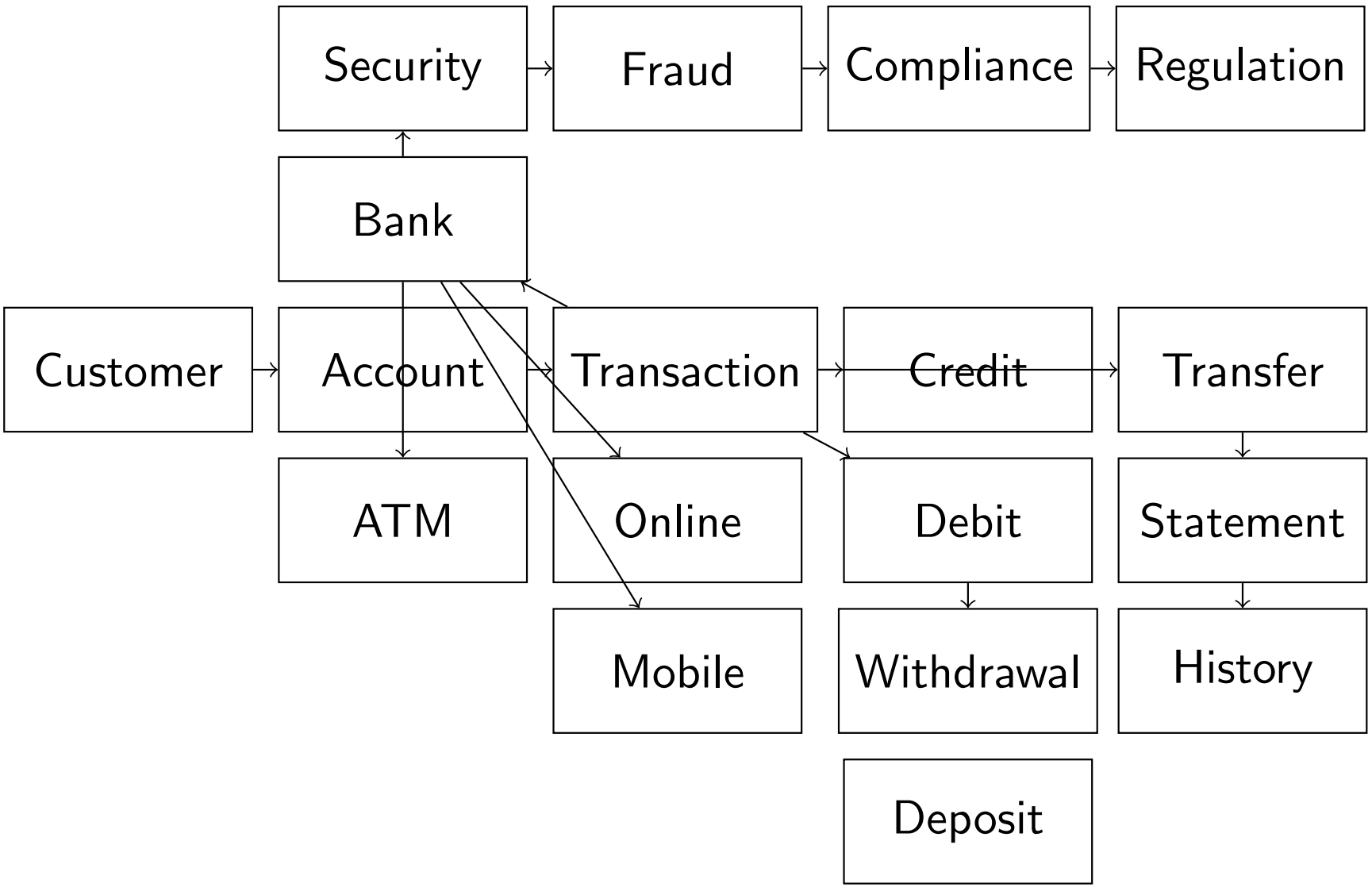
Example:



Modelling a Real-World Use Case



Modelling a Real-World Use Case



Core Concepts of OOP

- **Encapsulation:** Binding data and methods together to restrict access to an object's internal state.
- **Inheritance:** Deriving new classes from existing ones to promote code reuse.
- **Polymorphism:** Using a single interface to represent different data types and operations.
- **Abstraction:** Hiding the complex implementation details and exposing only the necessary functionality.

Encapsulation: Detailed Explanation

- Encapsulation ensures that an object's internal state is hidden from the outside world and can only be modified via specific methods (getters and setters).
- This helps in protecting the integrity of the object's data and preventing unintended interference.
- Example: A ‘Car‘ class might have private attributes like ‘speed‘ and ‘fuel‘, and public methods like ‘accelerate()‘ and ‘brake()‘ to manipulate them safely.

Example: Encapsulation in Java

```java
public class Student {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Listing 3: Encapsulation Example

Food for Thought: Encapsulation

- Can you think of a situation in your daily life where encapsulation is beneficial (e.g., personal information, financial data)?
- How can you apply encapsulation in a larger software project, like a web application?

Inheritance: Detailed Explanation

- Inheritance allows new classes to inherit properties and behaviors (methods) of existing classes.
- It promotes code reuse and establishes relationships between classes.
- Example: A ‘Vehicle‘ class could be extended to create specialized classes like ‘Car‘ or ‘Truck‘, which inherit general attributes like ‘speed‘ and ‘fuel‘ from ‘Vehicle‘, but also have their own specific attributes.

Example: Inheritance in Java

```java
public class Vehicle {
    protected String model;
    protected int speed;

    public void accelerate() {
        speed += 10;
    }
}

public class Car extends Vehicle {
    private int passengers;

    public void honk() {
        System.out.println("Honk!");
    }
}
```

Listing 4: Inheritance Example

Polymorphism: Detailed Explanation

- Polymorphism allows a single method or interface to be used for different data types or objects.
- It enables flexibility in code by allowing one interface to be implemented by multiple classes, each with its own behavior.
- Example: A ‘Shape‘ interface can have a method ‘draw()‘, and classes ‘Circle‘, ‘Rectangle‘, and ‘Triangle‘ can implement ‘draw()‘ differently for each shape.

Example: Polymorphism in Java

```
public interface Shape {
    void draw();
}

public class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

public class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}
```

Listing 5: Polymorphism Example

Abstraction: Detailed Explanation

- Abstraction focuses on exposing only the essential features of an object while hiding the implementation details.
- It helps reduce complexity and allows for simpler interaction with objects.
- Example: A ‘RemoteControl‘ class abstracts the complexity of controlling a ‘TV‘. The user just presses buttons without needing to understand the inner workings of the TV.

Advantages of OOP

- **Improved Code Maintainability and Reusability:** Classes and objects can be reused across different projects, saving time and effort.
- **Real-World Problem Modeling:** OOP mirrors real-world scenarios, making it easier to model complex systems.
- **Scalability and Ease of Debugging:** Code is modular and easy to extend. Bugs can be localized within specific objects.

Real-World Example: Banking Application

- **Encapsulation:** Account details stored privately and accessible through public methods.
- **Inheritance:** SavingsAccount and CheckingAccount are subclasses of Account.
- **Polymorphism:** A common ‘processTransaction()‘ method for different account types.

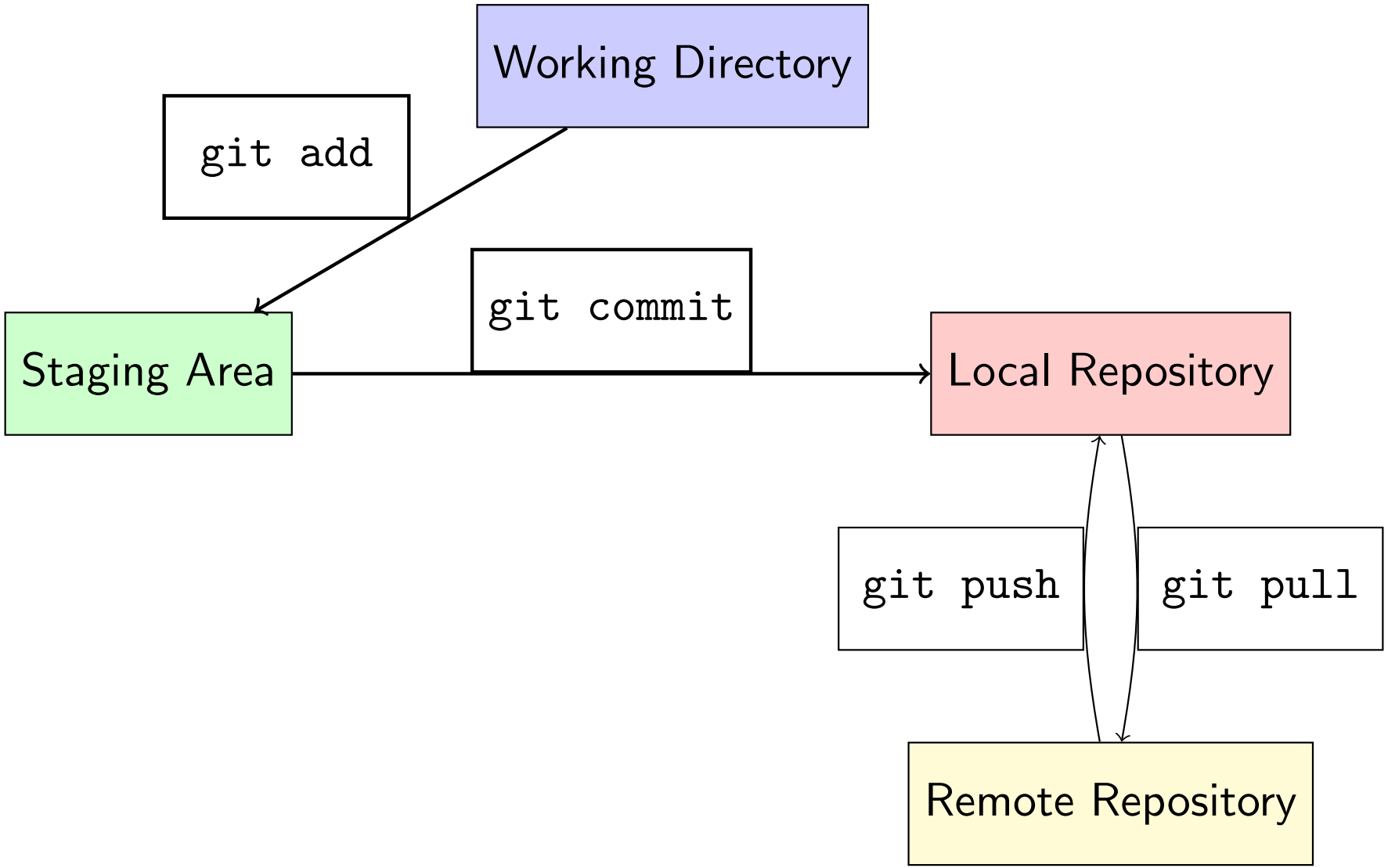
Outline

Basic Git Commands I

Essential Commands:

- `git init` - Initialize a new Git repository.
- `git clone <repo-url>` - Clone a remote repository.
- `git add <file>` - Stage changes for commit.
- `git commit -m "message"` - Commit staged changes with a message.
- `git status` - Show the working tree status.
- `git log` - View commit history.
- `git push` - Upload local commits to a remote repository.
- `git pull` - Fetch and integrate changes from the remote repository.

Basic Git Commands II

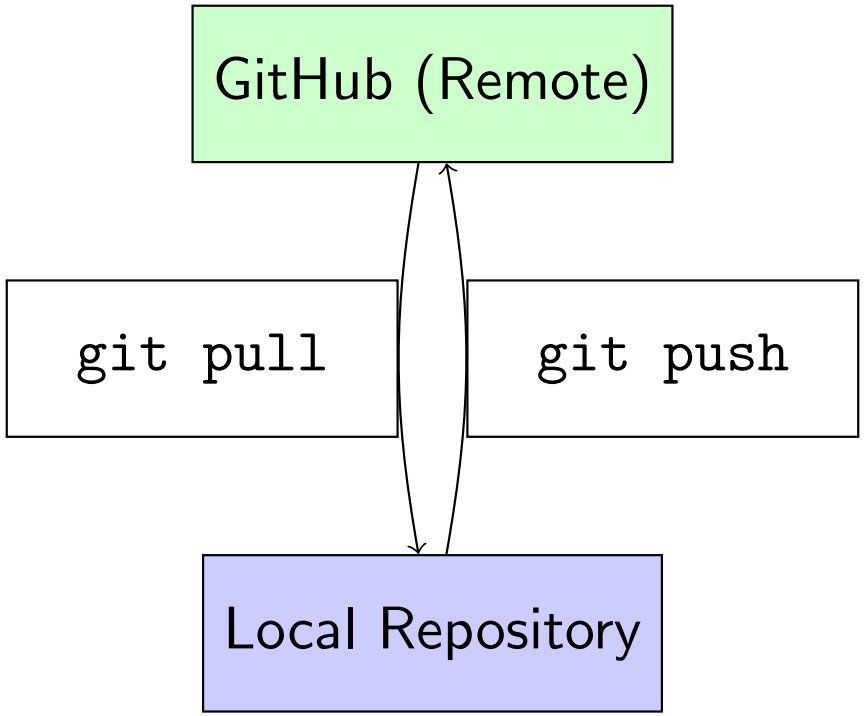


Connecting with GitHub I

Steps to Connect:

1. Create a GitHub account at `https://github.com`.
2. Generate an SSH key using `ssh-keygen` (Linux/Mac) or Git Bash (Windows).
3. Add the SSH key to your GitHub account under **Settings** $\rightarrow$ **SSH and GPG keys**.
4. Test the connection with `ssh -T git@github.com`.
5. Link a local repository:
   - `git remote add origin <repo-url>`.
   - `git push -u origin main`.

Connecting with GitHub II



OOP Evolution and Impact

- **Influence on Modern Programming:**
  - Frameworks: Spring, Django, Angular.
  - Design Patterns: Singleton, Factory, Observer.
- **Benefits:** Reusability, Modularity, Maintainability.
- **Limitations:** Overhead in small applications.

Common Pitfalls in OOP

- **Overusing Inheritance:** Leads to tight coupling.
- **Ignoring Encapsulation:** Exposing internal state directly.
- **Complex Hierarchies:** Difficult to manage and extend.

Common Pitfalls in OOP

- **Overusing Inheritance:** Leads to tight coupling.
- **Ignoring Encapsulation:** Exposing internal state directly.
- **Complex Hierarchies:** Difficult to manage and extend.

Solution: Prefer composition over inheritance, follow design principles.

Discussion Questions

1. Why is version control essential in collaborative development?
2. What is the impact of IDE customization on productivity?
3. How do design principles influence code quality?
4. Can you think of scenarios where procedural programming might outperform OOP?

Discussion Questions

1. Why is version control essential in collaborative development?
2. What is the impact of IDE customization on productivity?
3. How do design principles influence code quality?
4. Can you think of scenarios where procedural programming might outperform OOP?

OOP Concepts: Examples

Encapsulation Example:

- A ‘BankAccount‘ class with private attributes ‘balance‘ and public methods ‘deposit‘ and ‘withdraw‘ ensures controlled access.

Inheritance Example:

- A ‘Vehicle‘ class is extended by ‘Car‘ and ‘Truck‘ classes, inheriting attributes like ‘speed‘ while adding specific features.

Polymorphism Example:

- A ‘Shape‘ interface implemented by ‘Circle‘, ‘Rectangle‘, and ‘Triangle‘, each with its own ‘draw‘ method.

Achieving Reusability, Modularity, and Maintainability

- **Reusability:** Classes like ‘Logger‘ can be reused across different applications.
- **Modularity:** Separating functionality into independent modules, e.g., ‘PaymentProcessor‘ and ‘InventoryManager‘ in e-commerce.
- **Maintainability:** Encapsulation and abstraction reduce the risk of unintended changes affecting the entire system.

SOLID Principles in OOP

- **Single Responsibility Principle (SRP):** A class should have one and only one reason to change.
- **Open-Closed Principle (OCP):** Software entities should be open for extension but closed for modification.
- **Liskov Substitution Principle (LSP):** Derived classes must be substitutable for their base classes.
- **Interface Segregation Principle (ISP):** No client should be forced to depend on methods it does not use.
- **Dependency Inversion Principle (DIP):** Depend on abstractions, not on concretions.

DRY: Don't Repeat Yourself

- **Definition:** Avoid duplicating logic by abstracting common functionality into reusable components.
- **Benefits:** Easier maintenance, reduced risk of inconsistencies.
- **Example:** Extract common code for user authentication into a shared ‘AuthManager‘ class.

Outline

Exercises for Students

1. **Define Classes and Objects:**
   - Create a class ‘Book‘ with attributes like ‘title‘, ‘author‘, and ‘price‘.
   - Write methods to get and set these attributes.
   - Extend this class to include functionality such as calculating discounts based on the price.

2. **OOP in Practice:**
   - Implement a ‘Bank‘ class with methods for deposit and withdrawal.
   - Extend this to include different types of accounts using inheritance.
   - Add features like transaction history and interest calculation.

Discussion Questions

- Why is OOP considered more scalable compared to procedural programming?
- Can you think of a real-world scenario where polymorphism is useful?
- What are the trade-offs of using OOP versus other paradigms?

Outline

Recommended Resources: Books and Articles

- **Book: "Effective Java" by Joshua Bloch**
  - A comprehensive guide to writing good Java code with best practices.
  - Focuses on design patterns, Java-specific tips, and writing clean, maintainable code.
- **Article: "Object-Oriented Design & Programming" by Bjarne Stroustrup**
  - An insightful article by the creator of C++ that discusses the principles of object-oriented design.
- **Website: Oracle Java Tutorials**
  - Comprehensive tutorials directly from the creators of Java, covering OOP concepts in Java.
  - URL: `https://docs.oracle.com/javase/tutorial/`