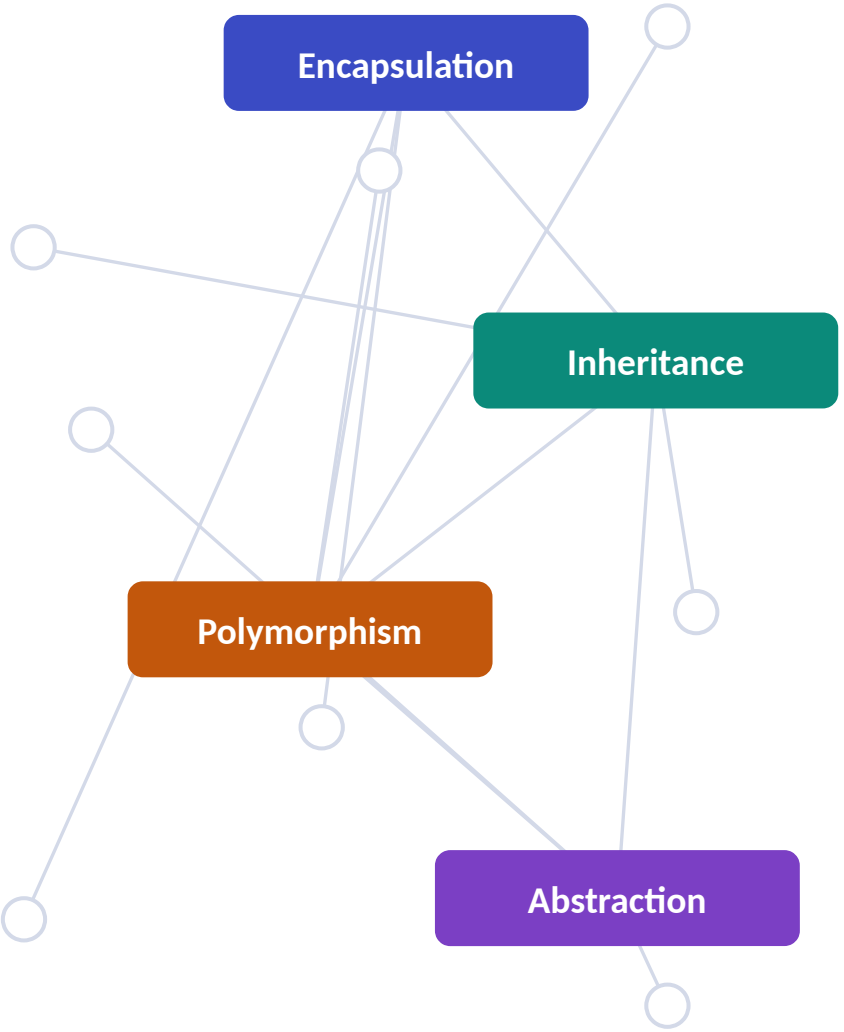
CSE 1115 · OBJECT-ORIENTED PROGRAMMING

The Four Fundamental Principles of Object-Oriented Programming

Understanding the ideas behind Encapsulation, Inheritance, Polymorphism, and Abstraction.



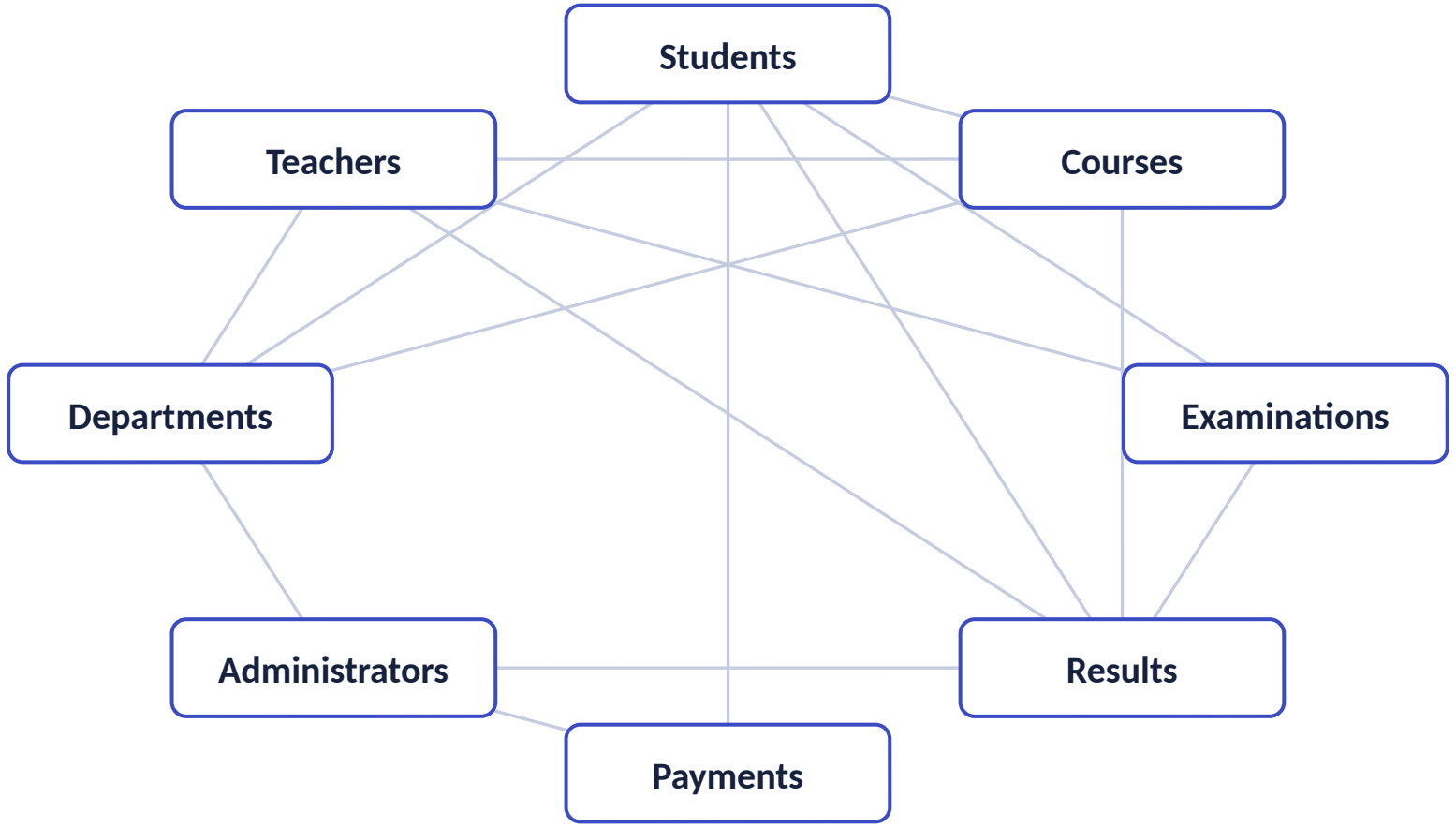
Sheikh Tanvir Ahmed

Lecturer, Department of Computer Science and Engineering

United International University

What happens when software becomes too complicated to manage?

A HYPOTHETICAL UNIVERSITY SYSTEM



THINK ABOUT IT

1. Who should access which information?
2. How can we avoid repeating the same code?
3. What if different objects must do similar things differently?
4. How can we hide complicated internal processes?

Why do we need OOP principles?

Working code is not enough.

Real software must organize complexity: many objects, relationships, rules and responsibilities.

1. **Controlled access**
   How do we protect an object's internal data?
2. **Code reusability**
   How do we avoid repeating common attributes and behaviors?
3. **Flexible behavior**
   How can different objects respond differently to the same operation?
4. **Managing complexity**
   How do we expose what is essential and hide the rest?

The four principles at a glance

1 Encapsulation

Protecting and controlling access to internal data.

THE PROBLEM IT ADDRESSES

Anyone could change an object's data.

2 Inheritance

Reusing and extending existing structures.

THE PROBLEM IT ADDRESSES

The same code is repeated across classes.

3 Polymorphism

Allowing the same operation to take different forms.

THE PROBLEM IT ADDRESSES

Similar objects must behave differently.

4 Abstraction

Hiding implementation complexity and exposing essential functionality.

THE PROBLEM IT ADDRESSES

Users drown in internal details.

These are OOP concepts, not Java keywords.

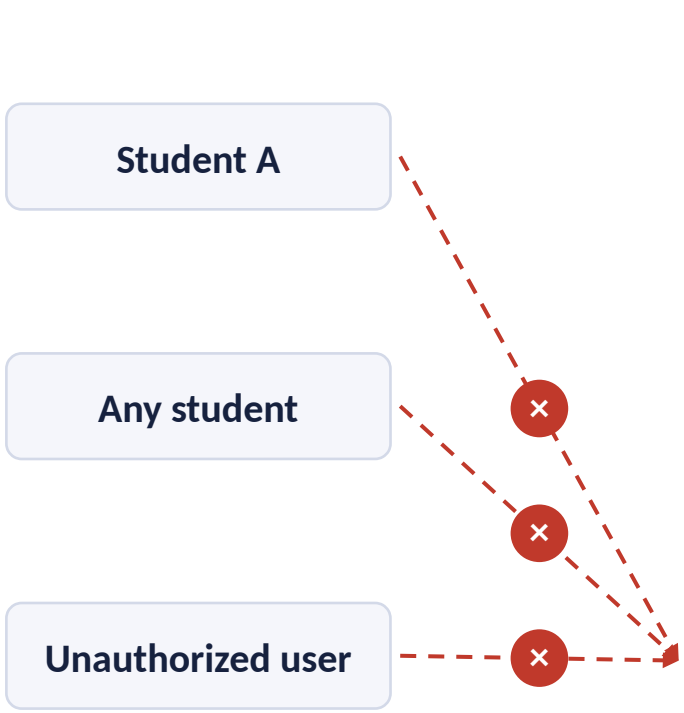
Concept vs. implementation

OOP PRINCIPLE · THE IDEA	JAVA MECHANISM · THE TOOLS
Encapsulation →	`private` fields, access modifiers, controlled methods
Inheritance →	`extends`
Polymorphism →	method overloading and overriding
Abstraction →	`abstract` classes and `interface`

The concept tells us **WHAT** we want to achieve. The language provides **HOW** we achieve it.

Other OOP languages, such as C++, use their own syntax for the same ideas. Our course focuses on Java.

The problem: should everyone see and change everyone's CGPA?



B

Student B

student profile

Name	Rahim Uddin
Student ID	011 221 045
Department	CSE
Email	rahim@example.edu

RESTRICTED

CGPA **3.82**

Should Student A change Student B's CGPA? **No**

Should every student read everyone's academic records? **No**

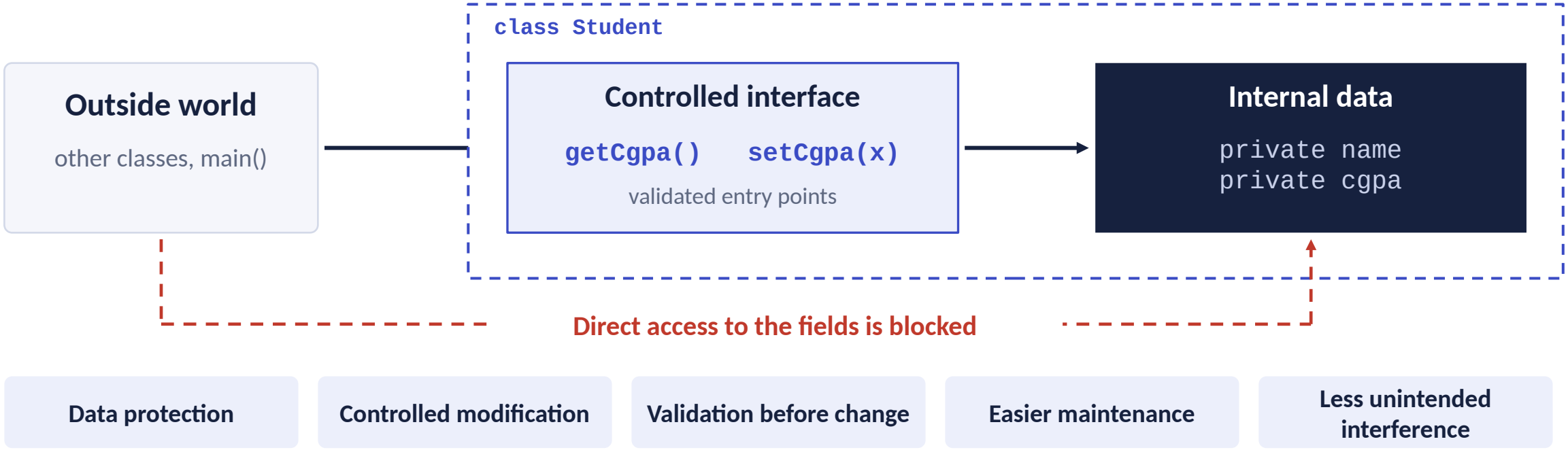
Should an unauthorized person edit a CGPA? **No**

> An object's internal information should not be **freely accessible or modifiable** by everyone.

> Java's `private` blocks direct access from other classes. A real system also needs authentication and permission checks; `private` alone is not security.

The solution: encapsulation

> **Encapsulation** bundles data and the methods that operate on it in one unit, while **controlling access** to the object's internal state.



Data protection	Controlled modification	Validation before change	Easier maintenance	Less unintended interference

Encapsulation is more than "make every field private": it is the design of the interface an object offers.

How does Java implement encapsulation?

	Same class	Same package	Subclass, other package	Everywhere
`private`	✓	–	–	–
`(no keyword)` package-private	✓	✓	–	–
`protected`	✓	✓	✓	–
`public`	✓	✓	✓	✓

Our focus: `private` fields and controlled `public` methods. A `private` superclass member is not directly accessible in a subclass; `protected` has extra cross-package rules.

Getters and setters

Getter
reads a value

Setter
changes a value through a checked method

Not every field needs both.

A value can be read-only, or changed only by a specific operation, such as processing results, instead of an arbitrary setter.

A public getter is not role-based access control.

Java code: implementing encapsulation

Student.java

```java
class Student {
    private String name;
    private double cgpa;

    public Student(String name, double cgpa) {
        this.name = name;
        setCgpa(cgpa);
    }

    public String getName() { return name; }
    public double getCgpa() { return cgpa; }

    public void setCgpa(double cgpa) {
        if (cgpa < 0.0 || cgpa > 4.0) {
            throw new IllegalArgumentException(
                "CGPA must be 0.0 to 4.0");
        }
        this.cgpa = cgpa;
    }
}
```

1

`private` fields

Other classes cannot touch name or cgpa directly.

2

Constructor reuses the setter

Validation applies from the moment the object exists.

3

`getCgpa()`: controlled read

Read access only. Anyone holding a Student can call it.

4

`setCgpa()`: validate first

Invalid values are rejected; the state stays valid.

> Validation is only a simple example. A real academic system must also control **who** may call an update.

Encapsulation: before vs. after

WITHOUT ENCAPSULATION

```
class Student {
    public double cgpa;
}
```

Nothing stops invalid data. Any code, anywhere, can break the object's state.

Another class runs:

```
student.cgpa = 10.0;
```

cgpa is now 10.0

CHECK YOUR UNDERSTANDING

Why is making every variable public a bad design decision?

WITH ENCAPSULATION

```
class Student {
    private double cgpa;

    public void setCgpa(double cgpa) {
        if (cgpa < 0.0 || cgpa > 4.0) {
            throw new IllegalArgumentException(
                "Invalid CGPA");
        }
        this.cgpa = cgpa;
    }
}
```

Another class runs:

```
student.setCgpa(10.0);
```

rejected, state stays valid

The problem: should every animal class rewrite eat() and sleep()?

Dog	Cat	Cow	Lion
`name`	`name`	`name`	`name`
`age`	`age`	`age`	`age`
`numberOfLegs`	`numberOfLegs`	`numberOfLegs`	`numberOfLegs`
`eat()`	`eat()`	`eat()`	`eat()`
`sleep()`	`sleep()`	`sleep()`	`sleep()`
`move()`	`move()`	`move()`	`move()`
copy-pasted	copy-pasted	copy-pasted	copy-pasted
`bark()`	**`meow()`**	**`moo()`**	**`roar()`**

DRY: Don't Repeat Yourself

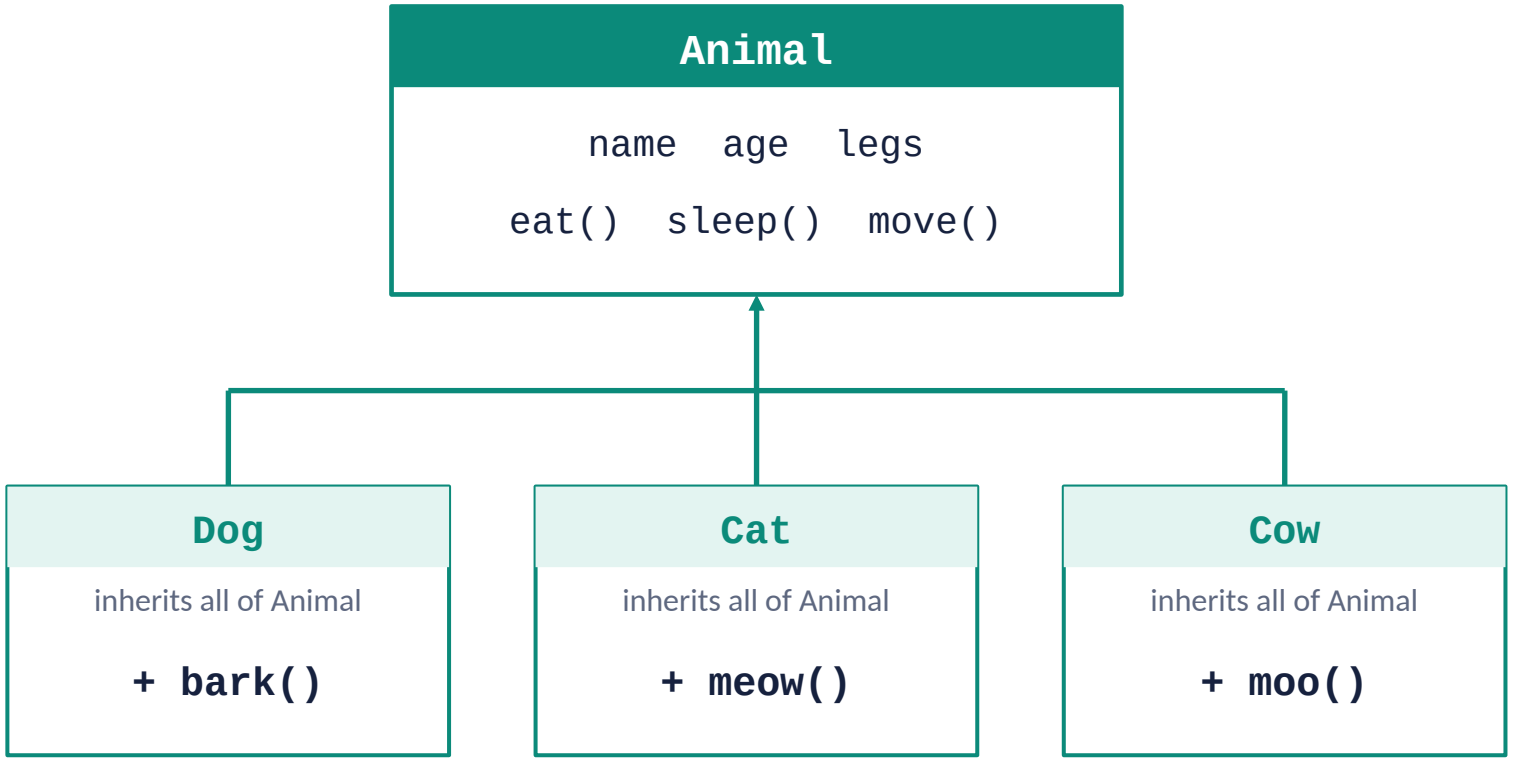
Every piece of knowledge should live in one place.

Repeated code is repeated maintenance

If `eat()` changes, we must edit Dog, Cat, Cow and Lion, and not miss one.

The solution: inheritance

> **Inheritance** lets one class acquire the accessible properties and behaviors of another, then **extend or specialize** them.



THE IS-A TEST

`Dog` is an Animal

`Cat` is an Animal

`Cow` is an Animal

Parent = Superclass = Base class

Child = Subclass = Derived class

Common parts live once, in the parent.

How does Java implement inheritance? The extends keyword

PARENT CLASS

Animal.java

```
class Animal {
    String name;

    void eat() {
        System.out.println("Animal is eating");
    }

    void sleep() {
        System.out.println("Animal is sleeping");
    }
}
```

CHILD CLASSES

Dog.java · Cat.java

```
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing");
    }
}
```

PARENT

Animal defines the shared members once.

INHERITED

name, eat(), sleep() come free with extends.

SPECIALIZED

bark() and meow() belong to one child only.

Reusing the parent class

Main.java

```java
public class Main {
    public static void main(String[] args) {

        Dog dog = new Dog();

        dog.name = "Tommy";

        dog.eat();
        dog.sleep();
        dog.bark();
    }
}
```

OUTPUT

```
Animal is eating
Animal is sleeping
Dog is barking
```

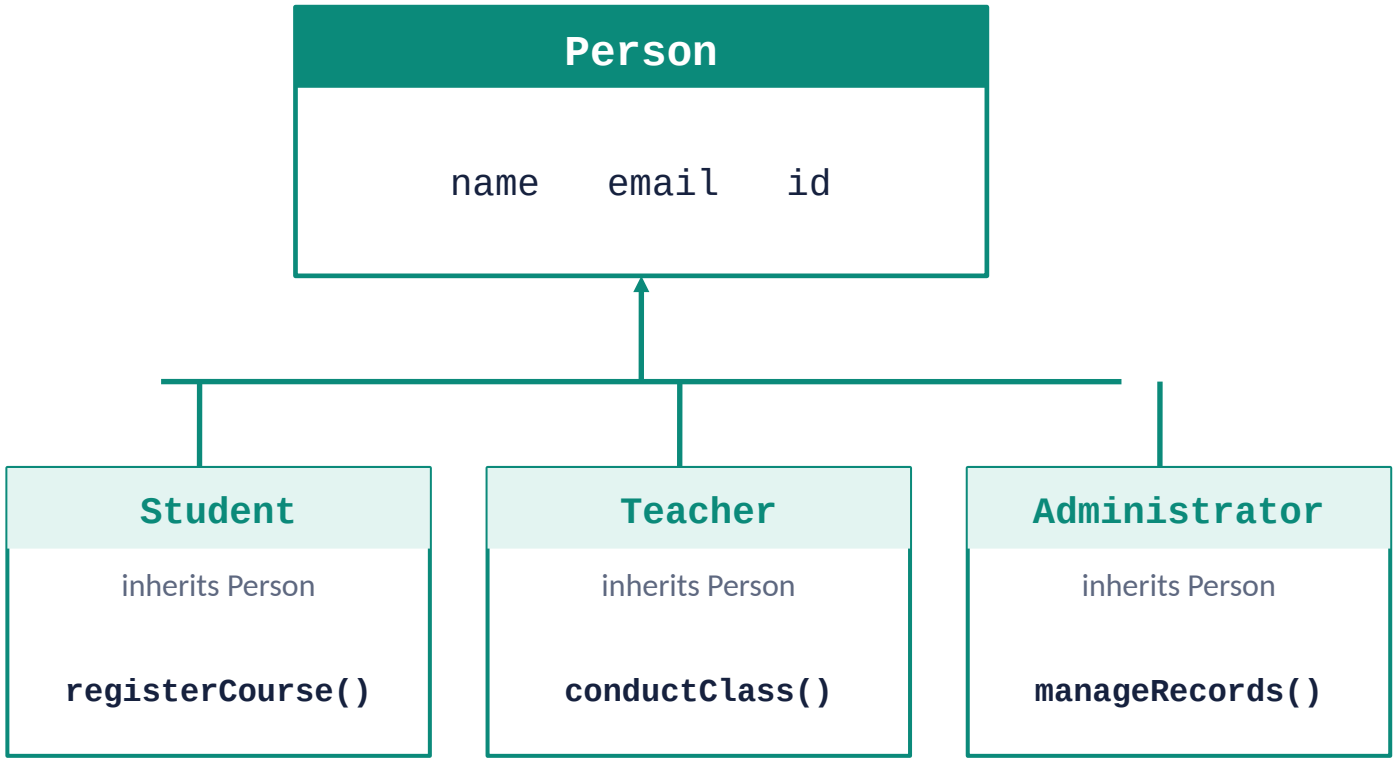
1. **Inherited**
   eat() and sleep() are written once in Animal.
2. **Own method**
   bark() exists only in Dog.

We never redefine `eat()` and `sleep()` in Dog. That is the advantage of inheritance.

name is package-private, so Main in the same package can access it.

Java class inheritance is **single**: a class extends only one class, but may implement many interfaces.

Inheritance is more than code reuse: it models relationships



IS-A, OR SOMETHING ELSE?

- ✓ **Student is a Person**
- ✓ **Dog is an Animal**
- × **Car is an Engine**

A Car **has** an Engine, so use composition: a field, not `extends`.

Do not inherit just because two classes happen to share a few variables.

CHECK YOUR UNDERSTANDING

Is a Dog an Animal? What about a Car and an Engine?

The problem: same purpose, different behaviors

Common goal: prepare for the examination `prepareForExam()`

CSE student

- Practices programming
- Solves algorithms
- Studies data structures

EEE student

- Solves circuit problems
- Studies electrical systems
- Practices numerical problems

Civil student

- Studies structural concepts
- Solves engineering calculations
- Studies construction materials

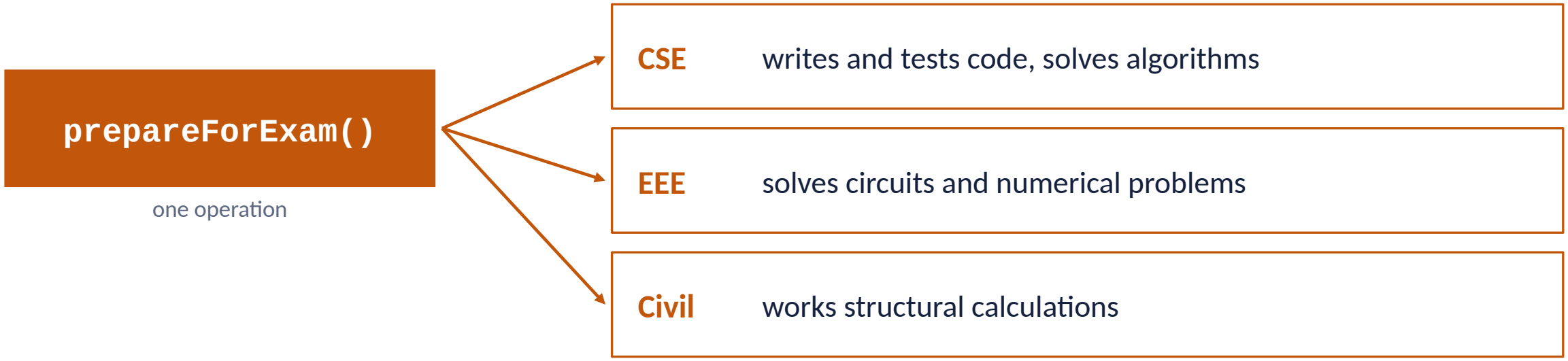
IDEA TO HOLD

One method name, three different implementations. Can we model that?

The solution: polymorphism

Poly = many · **Morph** = forms

> **Polymorphism** lets the same interface, operation, or method name represent **different behaviors** or forms.



Same operation ≠ same implementation

Two ways Java chooses which method runs

COMPILE-TIME

Method overloading

The compiler selects the overloaded method from the method name, the parameter list and the compile-time argument types.

RUNTIME

Overriding + dynamic dispatch

The JVM selects the overridden instance method from the actual class of the object, while the program runs.



Overloaded methods still **execute at runtime**; only the **selection** is made at compile time.

Compile-time polymorphism: method overloading

Method overloading: several methods in one class share a name but have **different parameter lists**.

Calculator.java

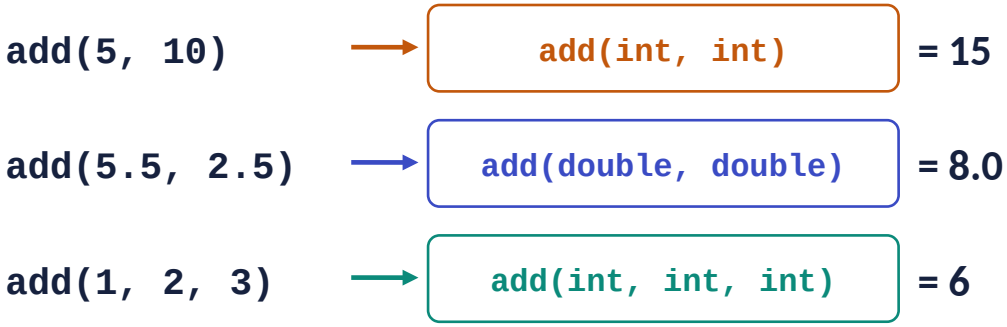
```java
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

```java
Calculator calc = new Calculator();

calc.add(5, 10);
calc.add(5.5, 2.5);
calc.add(1, 2, 3);
```



A different return type alone is NOT overloading.

```java
int add(int a, int b)
double add(int a, int b)   // compile error
```

Runtime polymorphism: method overriding

Animal.java · Dog.java · Cat.java

```java
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}
```

2

1

1

2 Inherited method

Animal defines a generic makeSound().

1 Overridden

Dog and Cat keep the same name and parameters, but replace the body.

! `@Override`

Asks the compiler to check that you really override.

Static methods are **hidden**, not overridden, so they do not give runtime polymorphism.

Runtime polymorphism: dynamic dispatch

PREDICT THE OUTPUT FIRST

Main.java

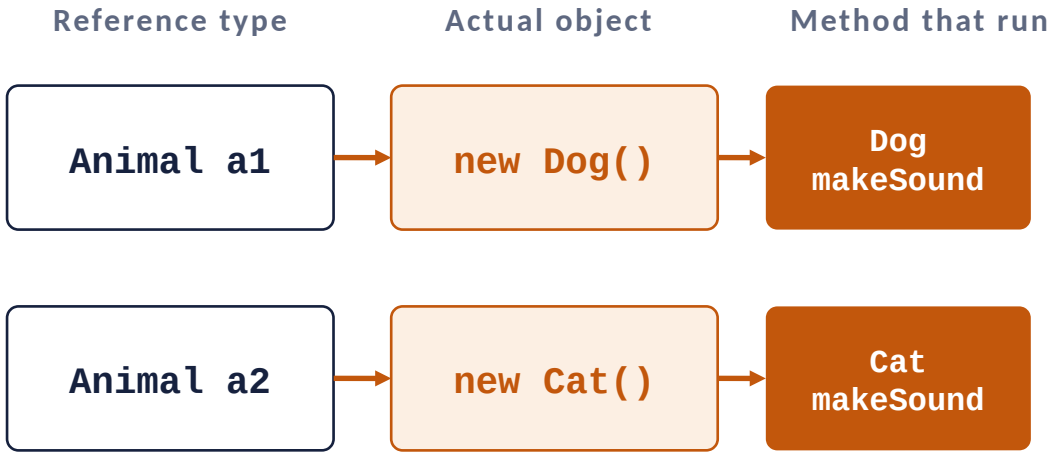
```java
public class Main {
    public static void main(String[] args) {

        Animal a1 = new Dog();
        Animal a2 = new Cat();

        a1.makeSound();
        a2.makeSound();
    }
}
```

OUTPUT

```
Dog barks
Cat meows
```



The reference type is **Animal**, but the **actual object** (Dog or Cat) decides which overridden method runs, at runtime. This is **dynamic method dispatch**.

extends → override → dynamic dispatch = runtime polymorphism

Overloading vs. overriding

Feature	Overloading	Overriding
Polymorphism type	Compile-time	Runtime
Method name	Same	Same
Parameter list	Different	Same
Inheritance required	No	Yes (class-based overriding)
Implementation	Separate versions by parameters	Specialized child version
Selection	Compile-time resolution	Runtime dynamic dispatch

CHALLENGE · OVERLOADING OR OVERRIDING?

A

```
class Printer {
    void print(int n) { }
    void print(String s) { }
}
```

B

```
class Dog extends Animal {
    @Override
    void makeSound() { }
}
```

Java has extra rules for static methods, access modifiers, return types and final methods. Here we focus on ordinary instance methods.

The problem: how does an ATM actually work?

What the user does

1. Insert the card
2. Enter the PIN
3. Select withdrawal
4. Enter the amount
5. Receive the money

simple interface

WHAT ACTUALLY HAPPENS BEHIND THE SCENES

Card validation	Contact the bank server	PIN verification
Account authentication	Balance verification	Transaction authorization
Balance update	Cash dispenser control	Transaction logging

ASK YOURSELF

Do we need to understand the entire internal implementation to use a system?

The solution: abstraction

Abstraction exposes essential functionality while hiding unnecessary implementation details: focus on **what** an operation does, not **how**.

ATM

EXPOSED · WHAT

Request a withdrawal

HIDDEN · HOW

Banking servers, authorization, cash control

DRIVING A CAR

EXPOSED · WHAT

Steering wheel, accelerator, brake, ignition

HIDDEN · HOW

Combustion and every engine component

ENCAPSULATION

Bundles state and behavior, and controls access to internal state.

≠

ABSTRACTION

Exposes essential functionality, hides implementation complexity.

How does Java implement abstraction?

ABSTRACT CLASS

Payment.java · alternative A

```
abstract class Payment {
    abstract void pay(double amount);

    void showMessage() {
        System.out.println("Processing...");
    }
}
```

- Declared with `abstract`; **cannot be instantiated** directly.
- Can mix abstract and concrete methods.
- Subclasses supply the abstract parts.

INTERFACE

Payment.java · alternative B

```
interface Payment {
    void pay(double amount);
}
```

- Declared with `interface`: a **contract** classes agree to follow.
- Classes use `implements`.
- A class can implement **multiple** interfaces.

> Abstraction is a **broader design principle**. Abstract classes and interfaces are Java's main tools for it, not its definition.

Java code: abstraction in action

Payment.java · BkashPayment.java · CardPayment.java

```java
interface Payment {
    void pay(double amount);
}

class BkashPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println(
            "Paid " + amount + " using Bkash");
    }
}

class CardPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println(
            "Paid " + amount + " using Card");
    }
}
```

Abstraction (the interface) and runtime polymorphism (the actual object chooses the behavior) work together.

Main.java

```java
public class Main {
    public static void main(String[] args) {

        Payment payment = new BkashPayment();

        payment.pay(500);
    }
}
```

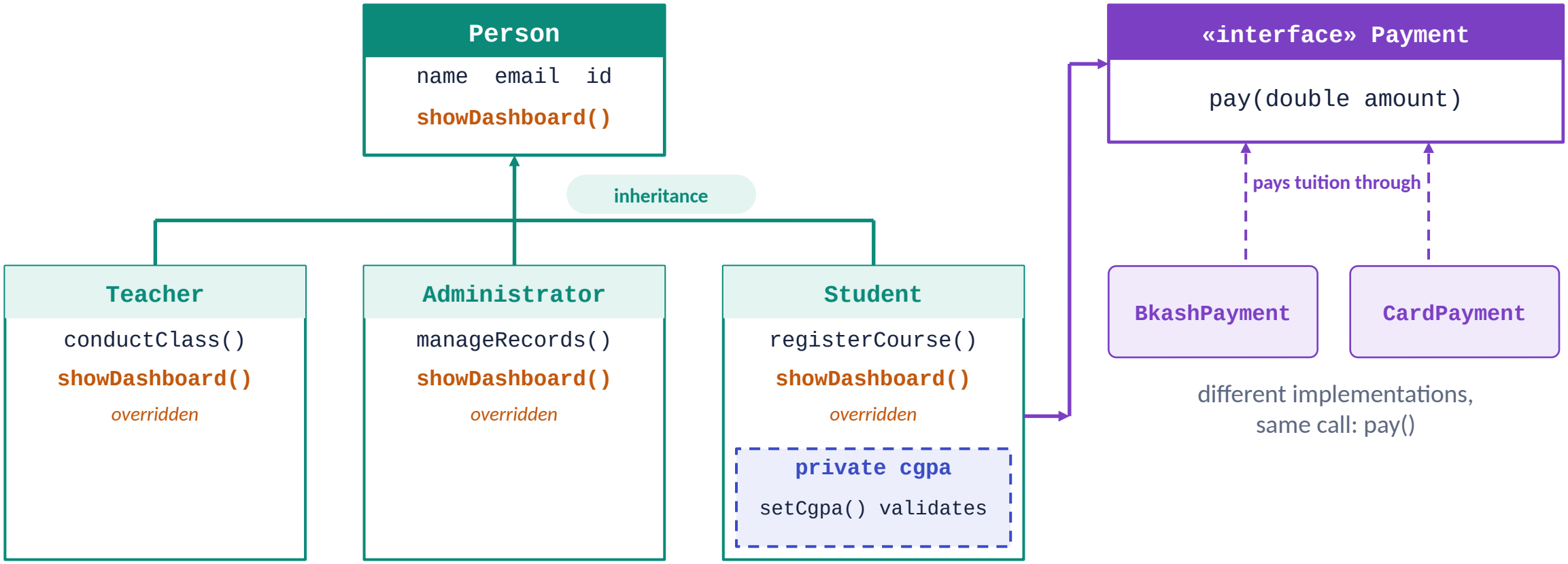
OUTPUT

```
Paid 500.0 using Bkash
```

1 **The user sees the contract**
Payment promises a pay() operation, not how it works.

2 **Each class supplies the how**
Swap in CardPayment and the same call behaves differently.

One system, four principles



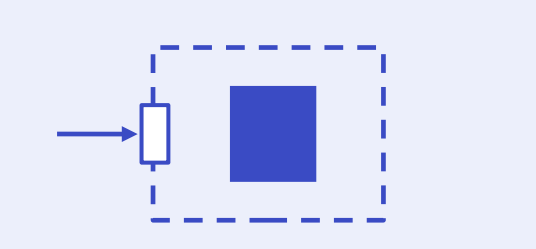
Encapsulation
CGPA changes only through validated methods.

Inheritance
Roles share common properties from Person.

Polymorphism
Each role does the same operation its own way.

Abstraction
Users see pay(), not the payment internals.

The four principles: final recap



Encapsulation

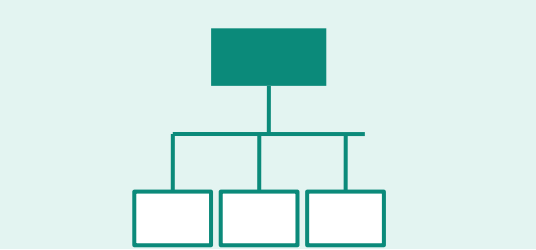
PROTECT

MAIN IDEA

Control access to internal state

JAVA MECHANISM

`private`, access modifiers, controlled methods



Inheritance

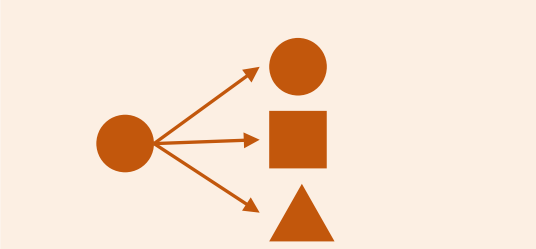
REUSE

MAIN IDEA

Reuse and extend existing classes

JAVA MECHANISM

`extends`



Polymorphism

DIFFERENT FORMS

MAIN IDEA

Same operation, different forms

JAVA MECHANISM

overloading, overriding



Abstraction

SIMPLIFY

MAIN IDEA

Expose essentials, hide implementation

JAVA MECHANISM

`abstract`, `interface`

Good OOP design is not about using more classes. It is about organizing complexity intelligently.