Féidearthachtaí as Cuimse
Infinite Possibilities

3 Encapsulation

Object Oriented programming through Java

OLLSCOIL TEICNEOLAÍOCHTA BHAILE ÁTHA CLIATH
TU DUBLIN
TECHNOLOGICAL UNIVERSITY DUBLIN

What we’ll cover

- **Revision**
- **Access Modifiers**
- **What** is encapsulation in OO
- **Why** is it so important?
- **How** do you implement it?

Revision

1. Creating an Object
In Java, when you want to use a class, you must **create an object** (an *instance* of that class):

```
Car myCar = new Car("Alice", "12-D-345");
```

Car → the class name.
myCar → the variable that refers to the object.
new Car(...) → calls the constructor to build the object.

Constructor Parameters

A constructor is a special method that runs when you create a **new** object.
It sets up the object’s starting values.
It can take parameters (values you pass in when creating the object).

```
public Car(String ownerName, String regNumber) {
    this.ownerName = ownerName;
    this.regNumber = regNumber;
}
```

String ownerName, String regNumber are constructor parameters.
They are **temporary variables** that only exist while the object is being created.
Whatever values you pass when using **new Car("Alice", "12-D-345")** get copied into the object.

Instance Variables

3Instance variables are the attributes that belong to each object.
They're declared at the top of the class, e.g.:
private String ownerName;
private String regNumber;

Each new object gets its *own copy*.
They live as long as the object exists.

When you do this:
Car myCar = new Car("Alice", "12-D-345");
"Alice" goes into the parameter ownerName.
"12-D-345" goes into the parameter regNumber.

The constructor copies them into the object's **instance variables**.
Now myCar has its own stored data:

```
    ownerName = "Alice"
    regNumber = "12-D-345"
```

Steps that happen when you create an object

Step 1: Call constructor

Arguments → "Alice", "12-D-345"

Step 2: Inside constructor

Parameters (temporary):

ownerName = "Alice"

regNumber = "12-D-345"

Step 3: Assign

Instance variables (permanent in object):

this.ownerName = "Alice"

this.regNumber = "12-D-345"

Step 4: Constructor ends

Parameters disappear

Instance variables remain in the object

ENCAPSULATION

Encapsulation “to enclose”

- Official definition:

Encapsulation is a way to bundle coding pieces together, allowing for greater **security** and **simplifying data hiding**.

- Can apply to
  - **attributes**
  - **methods**
  - classes

Encapsulation

- Can apply to
  - **attributes**
  - **methods**

Most common use

Example

I’m -1112 years old???

- `public` attributes

- Access from outside the class can’t be controlled

- Bad data

- Illustrates why **attributes** need encapsulation

Encapsulation for attributes

- `private` attribute

- `Public` getter / setter methods
  - For each attribute (usually)
  - Controlling access
  - Data is more secure

- And use setters from constructors !

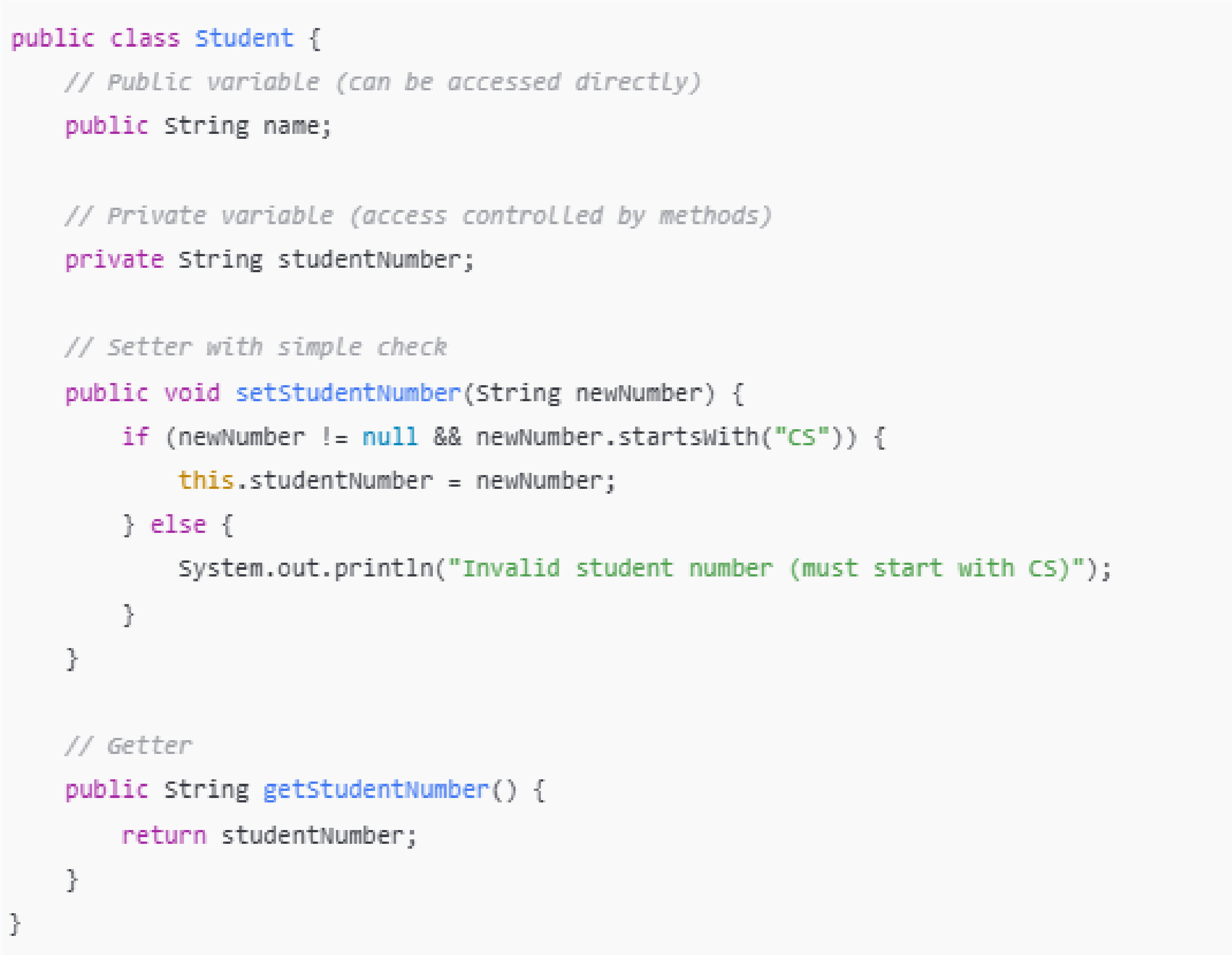
Encapsulated attribute

```
////
       private int age;

       ////  prevent bad data

       public void setAge(int newAge)
       {
         if (newAge >= 0 && newAge <=120)
            this.age = newAge;
         else
           System.out.println (“invalid age”); // or whatever error
                                                  handling  you want
       }
```

Another Example

```
public class Student {
    public String studentNumber;
      public String name;
// BAD: public
}
```

```
Student s1 = new Student();
s1.studentNumber = "12345";
s1.studentNumber = "banana";
// nonsense, but Java allows it
```



```java
public class Student {
    // Public variable (can be accessed directly)
    public String name;

    // Private variable (access controlled by methods)
    private String studentNumber;

    // Setter with simple check
    public void setStudentNumber(String newNumber) {
        if (newNumber != null && newNumber.startsWith("CS")) {
            this.studentNumber = newNumber;
        } else {
            System.out.println("Invalid student number (must start with CS)");
        }
    }

    // Getter
    public String getStudentNumber() {
        return studentNumber;
    }
}
```

Good Design

Java access modifiers

- Access modifiers **control visibility** of classes, variables, and methods.

- They help **enforce encapsulation**, a key principle in OOP.
- Java has **four** main access levels:
    - public
    - private
    - protected
    - (default) – package-private (no keyword)

Visibility level

Modifier	Visible To
public	EVERYWHERE
protected	Same package + subclasses
Default	Same package
private	Class

- Helps manage **who can see or modify** your data.

- Prevents misuse or unintended access.

Encapsulation

- Can apply to
  - **attributes**
  - **methods**

Encapsulation

- Can apply to
  - **attributes**
  - **methods**

- Sometimes, want to keep methods private

- Example
  - `ATM` class

Example

- Can apply to
  - **attributes**
  - **methods**

```
class BankAccount {
    private double balance;

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

    // Private method: Applies a fee on withdrawals
    private void applyTransactionFee() {
        balance -= 2.00; // Deduct a fixed fee
    }

    // Public method: Withdraw money while ensuring fee is applied
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            applyTransactionFee(); // Fee applied internally
            System.out.println("Withdrawal successful! New balance: $" + balance);
        } else {
            System.out.println("Insufficient funds or invalid amount.");
        }
    }
```

- **What's going on in this code?**
  What can you call or not from outside the class (e.g. main method)?

Advantages of Encapsulation?

- Data protection

- Controlled access

- Hides complexity and implementation details

Advantages of Encapsulation?

- **Data protection**

E,g, setters

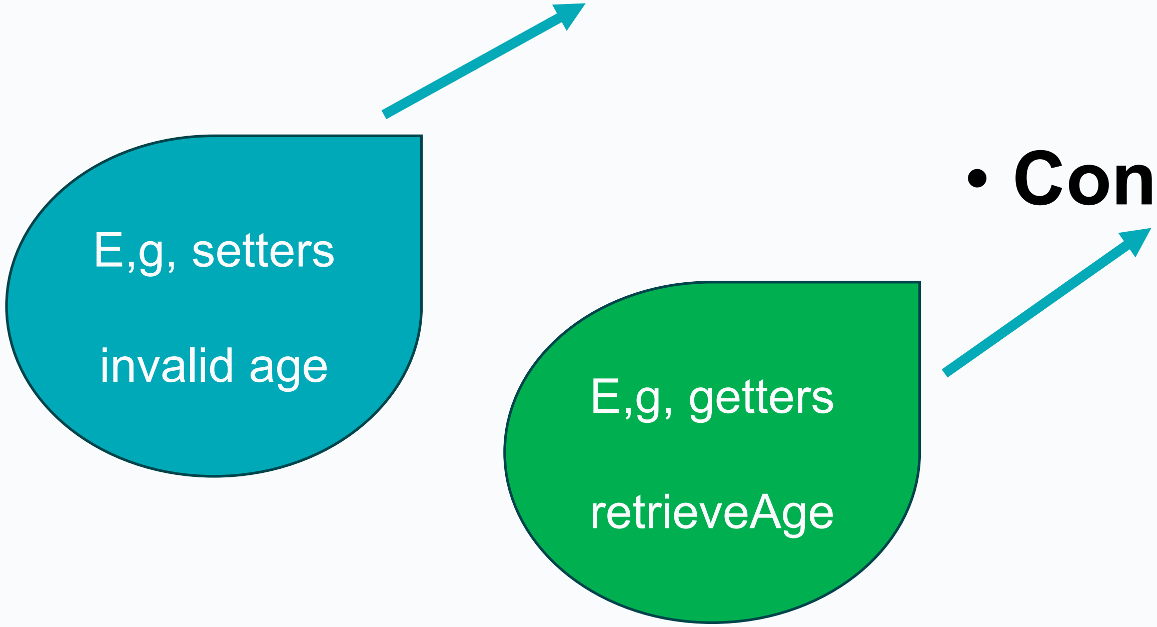
invalid age

- **Controlled access**

- **Hides complexity and implementation details**

Advantages of Encapsulation?

- **Data protection**



E,g, setters

invalid age

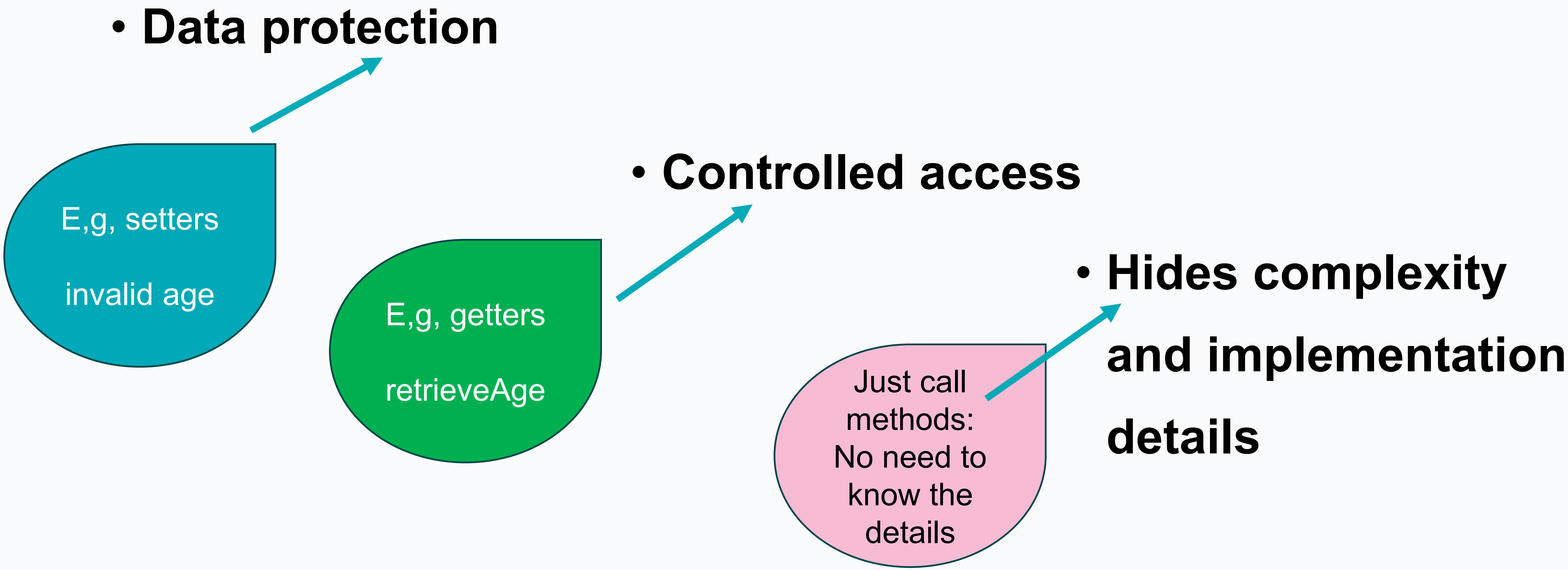
E,g, getters

retrieveAge

- **Controlled access**

- **Hides complexity and implementation details**

Advantages of Encapsulation?



- **Data protection**

E,g, setters

invalid age

- **Controlled access**

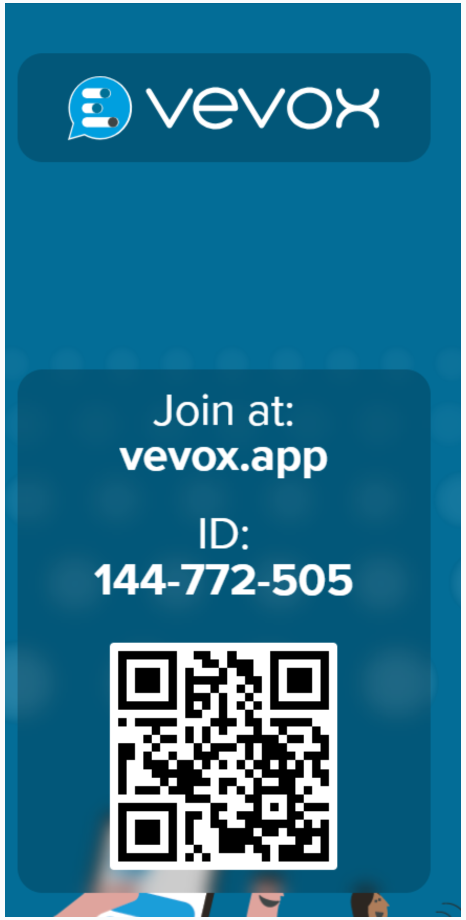
E,g, getters

retrieveAge

- **Hides complexity and implementation details**

Just call methods: No need to know the details

Test your Knowledge



vevox

Join at:
vevox.app

ID:
144-772-505

Test your knowledge

Which keyword is used to implement encapsulation in Java by restricting access to class members?

A) public
B) private
C) final
D) abstract

What will happen if you try to access a private field from outside its class?

A) It will work normally
B) It will cause a compilation error
C) The field will be automatically converted to public
D) Java will generate a default getter method

Which of the following correctly demonstrates encapsulation in Java?

A) Making all variables public and accessing them directly
B) Making variables private and providing getter and setter methods
C) Using static methods to modify class attributes
D) Allowing all methods to modify the class variables directly

Covered

- **What** is encapsulation in OO
- **Why** is it so important
- **How** do you implement it

- Some additional notes

Encapsulation example

```
public class Person
{
        private String name;
        private String streetAddress;
        private int age;
        // constructors go here.. But left out for this example
        // getter and setter methods needed per each of the 3 attributes)
        //example Setter method for “age” attribute
        public void setAge(int newAge)
        {
          if (newAge >= 0 && newAge <=120)
             this.age = newAge;
          else
            System.out.println (“invalid age”); // or whatever error
                                             handling  you want
        }
        //example getter method for “age” attribute
        public int getAge()
        {
            return this.age;
        }
```

*Un*encapsulated

- Basic class definition :

```
public class Person
{

// Example of unencapsulated attributes…bad..!!
      String name;
      String streetAddress;
      int age;

// constructor which initialises the instance
variables
      public Person(String name)
      {
             this.name = name;
      }
```

Problem/ solution

- Any code external to the class can change them!!
  - **E.g.from an external class:**

```
Person p1 = new Person (“Clara”);
p1.age = -12;   // nooooo!!!
```

- To prevent uncontrolled access to object data (i.e. to keep them “safe” using encapsulation):
  - Set attributes `private //Unavailable outside of the class`
  - Use a getter and setter method for each attribute