Java Encapsulation

Table of contents

Classes and objects 1
Encapsulation 1
Constructors 4
Access modifiers 5
Getters and setters 6
The `static` keyword 6
Class invariants 8
Method signatures 9

Introduced in 2026-03-05-object-oriented-programming-i[1] (Lecture, Week 2).

Classes and objects

- A **class** describes the contents of the objects that belong to it: an aggregate of data fields (properties) plus the operations (methods) defined on them.
- An **object** is an element (instance) of a class; objects have the behaviours of their class. The object is the actual component of a running program, while the class specifies how instances are created and how they behave.

Encapsulation

Encapsulation is the process of bundling code (methods/member functions) and data (member variables) together into a single unit — a class. It restricts direct access to some of an object's components, preventing the accidental modification of data.

The Java that box stands for:

[1] courses/csse2002/2026-03-05-object-oriented-programming-i.pdf

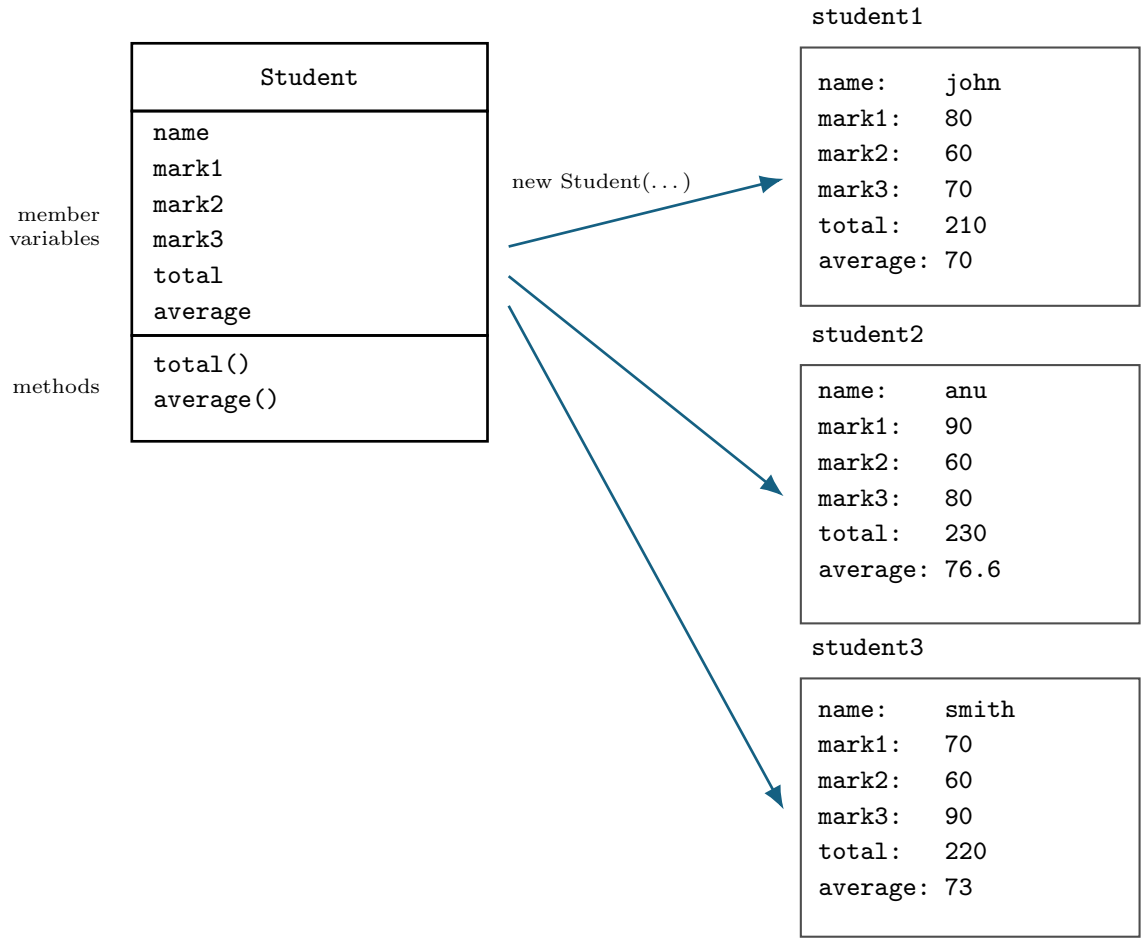

Figure 1: One Student class, three Student objects: the class fixes the member variables and methods, each object carries its own values for them. Each arrow is one new Student(…) call creating an object. anu's average is rounded to 76.6.

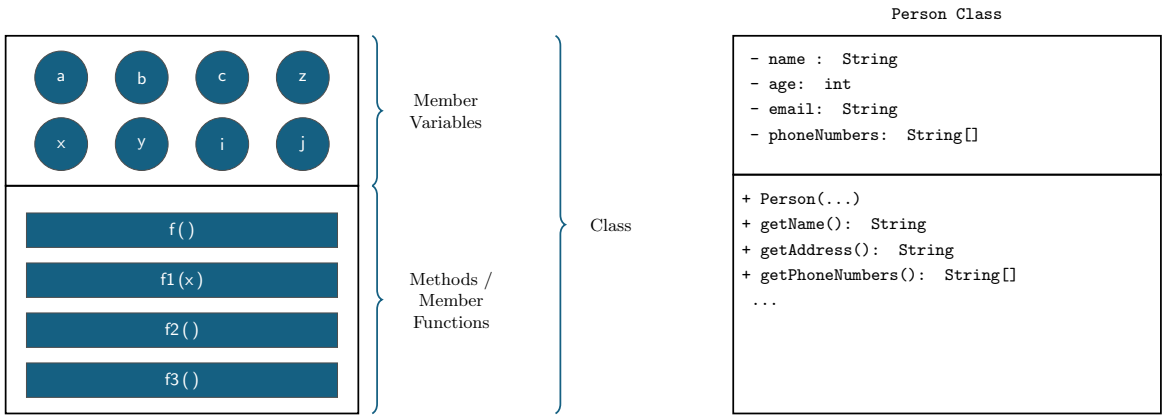

Figure 2: What a class is made of, and how it is written as a diagram. Left: a member-variable compartment above a methods compartment, the two braced together as the class. Right: the same idea as a two-compartment box, attributes above operations, with the class name above the box. The attribute list and the operation list in the box do not quite match (email against getAddress).

```java
public class Person {
    private String name;
    private int age;
    private String email;
    private String[] phoneNumber;

    public Person(String name, int age, String email, String[] phoneNumber) {
        this.name = name;
        this.age = age;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public String getEmail() { return email; }
    public String[] getPhoneNumber() { return phoneNumber; }
}
```

The box and the code do not match exactly: the box lists an `email` attribute but a `getAddress()` operation (the box has no `address` attribute and no `getEmail()` operation, while the code has `email` and `getEmail()` and no `address` at all), and the box spells the array field `phoneNumbers` where the code spells it `phoneNumber`. The intended mapping is the obvious one — one private attribute per `-` line, one public method per `+` line, the constructor in the operations compartment with no return type.

Constructors

A constructor is a special method invoked when an object of the class is created (via `new`):

```
public class Person {
    private String name;
    private int age;

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

    public static void main(String[] args) {
        Person person = new Person("John", 20);
        System.out.println("The name of the person is " + person.name);
    }
}
```

If a class has no constructor, the Java compiler automatically creates a **default constructor**. Take the same class with the constructor deleted and `new Person()` called instead — it still compiles, because the compiler supplies the no-argument constructor. `name` is never assigned, so the `println` would print `null`.

```
public class Person {
    private String name;
    private int age;

    public static void main(String[] args) {
        Person person = new Person();
        System.out.println("The name of the person is " + person.name);
    }
}
```

Notes on constructors:

- Constructors are invoked implicitly when objects are instantiated.
- The constructor's name MUST be the same as the class.
- A Java constructor must not have a return type.
- If a class does not have a constructor, the compiler creates a **default constructor**. This happens at compile time, not at run time: the compiler emits it into the class file.
- A constructor can be **overloaded** but can **not** be **overridden**.

Access modifiers

Access modifiers set the accessibility (visibility) of classes, interfaces, properties, methods, constructors, and data members:

Modifier	Accessible from
`private`	Only within the declaring class
`public`	Anywhere
`protected`	Same package, plus subclasses (see java-inheritance[2])
(default, no keyword)	Only within the same package

The same `BankAccount` under three of the four:

`private` — the field cannot be touched from outside the class, so this does not compile:

```java
public class BankAccount {
    private double balance;
    private String accountNumber;
    private String accountHolder;
    private double interestRate;
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.balance = 1000.0;   // compile error
    }
}
```

```
java: balance has private access in Week02.OOP.AccessModifers.BankAccount
```

`public` — change `balance` to `public double balance;` and the assignment that would not compile above is now legal. Add `System.out.println("Balance: " + account.balance);` to `Main` and it prints `Balance: 1000.0`.

default (no keyword) — drop the modifier entirely (`double balance;`) and, because `Main` is in the *same package*, it also compiles and prints `Balance: 1000.0`. The difference from `public` only shows up across a package boundary.

`protected` is **homework**: read DigitalOcean's Java access modifiers tutorial. It is accessible within the same package *as well as* from subclasses. The subclass clause is what makes `protected` wider than default access, because it reaches subclasses in *other* packages too.

[2] courses/csse2002/java-inheritance.pdf

Getters and setters

Getter and setter methods provide controlled access to an object's private fields, keeping its internal representation hidden from the outside world:

- **Getters** (accessors) retrieve the value of a private field from outside the class.
- **Setters** (mutators) set/update the value of a private field from outside the class.

```java
public class BankAccount {
    private double balance;
    private String accountNumber;

    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
    }

    public double getBalance() { return balance; }
    public void setBalance(double balance) { this.balance = balance; }
    public String getAccountNumber() { return accountNumber; }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("111000");
        account.setBalance(10000);
        System.out.println("Balance: " + account.getBalance());
        System.out.println("Account number: " + account.getAccountNumber());
    }
}
```

The example is asymmetric: `balance` gets both a getter and a setter, `accountNumber` gets only a getter. Whether a field is writable from outside is a per-field design decision, not something you owe every field.

The `static` keyword

`static` is a **non-access** modifier for methods and attributes:

- Static methods/attributes can be accessed without creating an object of the class.
- Useful in memory management; can be applied to variables, methods, blocks, and nested classes.
- In *procedural* programming the same keyword means something different: it marks a variable or function whose scope is limited to the file it is declared in.

Static variables (class variables) are shared among all instances of a class — useful for constants and shared properties:

```
class BankAccount {
    public static int totalAccounts = 0;

    private double balance;
    private String accountNumber;
    private String accountHolder;

    public BankAccount(double balance, String accountNumber, String accountHolder) {
        this.balance = balance;
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
    }

    public String getAccountNumber() { return accountNumber; }
    public double getBalance() { return balance; }
    public String getAccountHolder() { return accountHolder; }
}
```

Static methods can be called without creating an instance, but **cannot access non-static (instance) variables or methods directly**:

```
class Bank {
    static double interestRate = 5.0;

    static double calculateInterest(double balance, int years) {
        return (balance * interestRate * years) / 100;
    }
}

class BankAccount {
    private String accountHolder;
    private double balance;

    BankAccount(String name, double balance) {
        this.accountHolder = name;
        this.balance = balance;
    }

    void showInterest(int years) {
        double interest = Bank.calculateInterest(balance, years);
        System.out.println(accountHolder + " earns $" + interest + " in " + years + "
        ↪  years.");
    }
}
```

The call site is the point of the example: it reads `Bank.calculateInterest(...)` — on the *class*, not on an instance — and because `calculateInterest` is static it cannot reach any account's `balance` itself, so the balance has to be handed to it as an argument. The static field `interestRate`, by contrast, it can read directly, because that field belongs to the class too.

Class invariants

A class invariant specifies a condition (or set of conditions) that should always be true throughout the life of an object — used to ensure a system remains in a valid state. Put more precisely, it is a constraint that must hold **at every stable point in time** during the life of an object. A class invariant:

1. Must be established after the class constructor.
2. May be assumed as a precondition of each method (excluding the constructor).
3. Must be established after each method call.

```
class Counter {
    private int count;
    public Counter() { count = 0; }
    public void increment() { count = count + 1; }
}
// Invariant: count >= 0
```

Trace the three rules against `Counter`: the constructor sets `count = 0`, which establishes `count >= 0`; `increment` may *assume* `count >= 0` on entry; and since it only ever adds 1, `count >= 0` still holds on exit. Nothing else can reach `count`, because it is `private` — which is what makes the invariant enforceable at all rather than merely hoped for.

Invariants complement **preconditions** (what must be true before a method executes) and **postconditions** (what must be true after) — together these define a contract for how a method or function should behave. Pre/postconditions themselves are in java-specification[3] and java-hoare-logic[4]; see also week8-tutorial-class-invariants[5].

Protecting invariants

The `Counter` example does not show how an invariant can be broken from outside. Guarding against that is what makes the definition enforceable rather than merely stated; see also week8-tutorial-class-invariants[6] and the defensive-programming material in java-specification[7].

[3] courses/csse2002/java-specification.pdf
[4] courses/csse2002/java-hoare-logic.pdf
[5] courses/csse2002/week8-tutorial-class-invariants.pdf
[6] courses/csse2002/week8-tutorial-class-invariants.pdf
[7] courses/csse2002/java-specification.pdf

A specification can *claim* an invariant while the implementation still allows it to be broken. Two common leaks:

1. **Public fields** — a directly-mutable field lets any caller bypass the class entirely and violate the invariant. Fix: make the field `private`.
2. **Returning an internal reference** — a getter that `return`s its internal mutable collection/object directly hands the caller a way to mutate it from outside, bypassing any checks the class's own methods perform. Fix: return a **defensive copy**:

```
private List<String> files;

public List<String> getFiles() {
    return new ArrayList<>(files); // copy, not the internal reference
}
```

Preserving the invariant may also require adding **preconditions** to methods that could otherwise violate it (e.g. rejecting an addition that wouldn't satisfy the invariant), or, when a method is likely to be called by code outside your control, defensively throwing an exception (e.g. `IllegalArgumentException`/`IllegalStateException`) rather than relying purely on the precondition contract (see java-specification[8] — Defensive programming).

Method signatures

A method signature is a method's unique identifier: its **name** plus its **parameter types**. The return type and the parameter *names* do **not** count towards the signature, and a signature MUST be unique within a class:

```
public void print() { ... }              // signature: print()
public void print(int parameter) { ... } // signature: print(int)
```

Worked question — is this allowed?

```
public class Action {
    public Boot fish(int strength, int rodLength) { ... }
    public Fish fish(int strength, int polLength) { ... }
}
```

No. Both methods have the signature `fish(int, int)`. The differing return types (`Boot` vs `Fish`) and the differing parameter names (`rodLength` vs `polLength`) are exactly the two things a signature ignores, so the class will not compile. This same rule is what makes overriding

[8] courses/csse2002/java-specification.pdf

work in java-inheritance[9]: an override must repeat the superclass method's signature exactly, or it is a new overload rather than an override.

[9] courses/csse2002/java-inheritance.pdf