Java Encapsulation

Table of contents

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.

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

```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; }
}
```

Constructors

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

```java
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** at runtime.

Notes on constructors:

- Constructors are invoked implicitly when objects are instantiated.
- The constructor's name MUST be the same as the class.

- A constructor must not have a return type.
- A constructor can be overloaded but cannot 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

Attempting to access a `private` field from outside its class is a compile error (`X has private access in Y`).

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; }
}
```

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

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.

Static variables (class variables) are shared among all instances of a class — useful for constants and shared properties, e.g. `public static int totalAccounts = 0;`.

Static methods can be called without creating an instance of the class, 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 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. 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
```

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 should behave.

Protecting invariants

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[3] — Defensive programming).

Method signatures

A method signature is a method's unique identifier: its name plus its parameter types. The return type and 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)
```

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