Week 12
OOP - Encapsulation

ISB02303103 - Algorithm & Programming Language

Semester Gasal 2024/2025

4 sks

OOP

- ~~Basic OOP Concepts~~
- Encapsulation
- Inheritance

What is encapsulation?

The meaning of Encapsulation, is to make sure that **"sensitive" data** is **hidden** from users.

To achieve this, you must:

- declare class variables/attributes as **private**
- provide public **get** and **set** methods to **access and update** the value of a private variable

JAVA Modifier

The public keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- **Access Modifiers - controls the access level**
- Non-Access Modifiers - do not control access level, but provides other functionality

Access Modifiers

For classes, you can use either public or default:

public	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

For **attributes, methods and constructors**, you can use the one of the following:

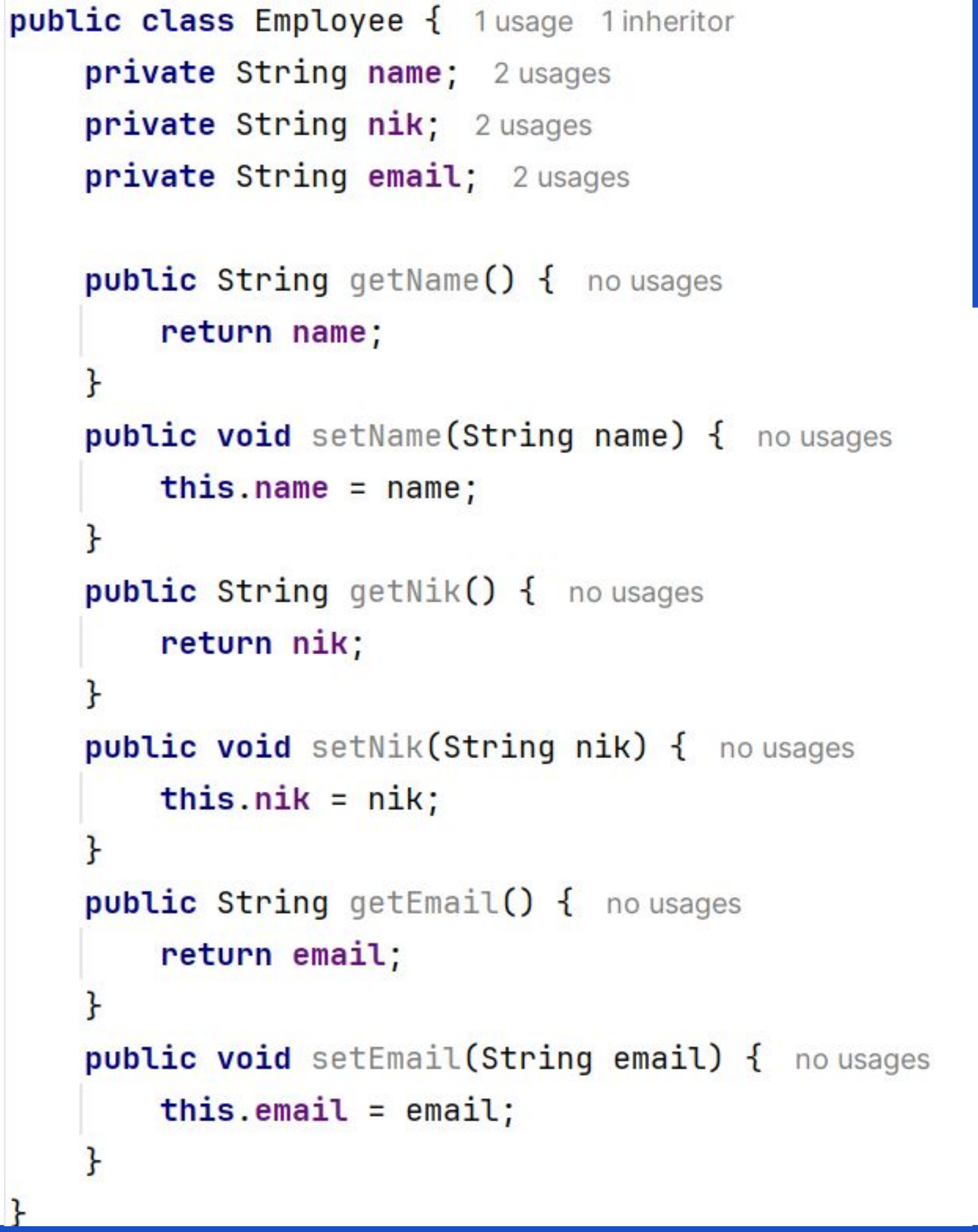
public	The code is accessible for all classes
private	The code is only accessible within the declared class
protected	The code is accessible in the same package and subclasses

Get and Set

Private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public get and set methods.

The **get** method **returns** the variable value, and the **set** method **sets** the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:



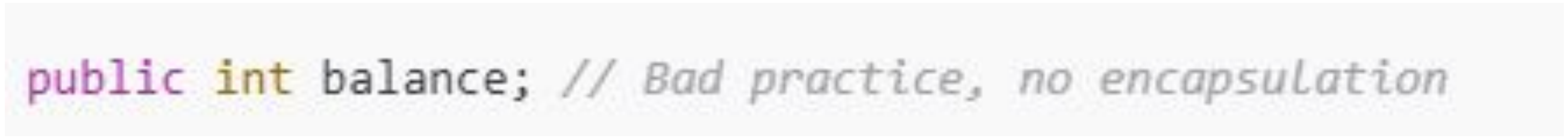
```
public class Employee {  1 usage  1 inheritor
    private String name;  2 usages
    private String nik;  2 usages
    private String email;  2 usages

    public String getName() {  no usages
        return name;
    }
    public void setName(String name) {  no usages
        this.name = name;
    }
    public String getNik() {  no usages
        return nik;
    }
    public void setNik(String nik) {  no usages
        this.nik = nik;
    }
    public String getEmail() {  no usages
        return email;
    }
    public void setEmail(String email) {  no usages
        this.email = email;
    }
}
```

Why Encapsulation?

- Better control of class attributes and methods
- Class attributes can be made read-only (if you only use the get method), or write-only (if you only use the set method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

Common Mistakes

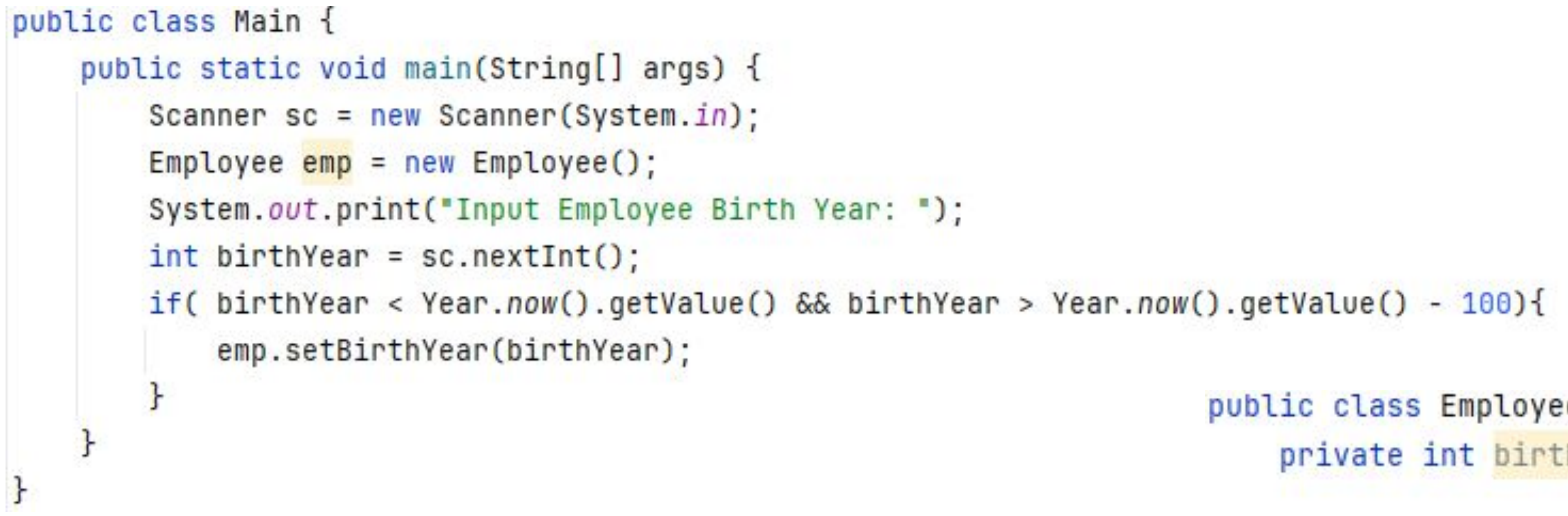
1. Direct Access to Variables



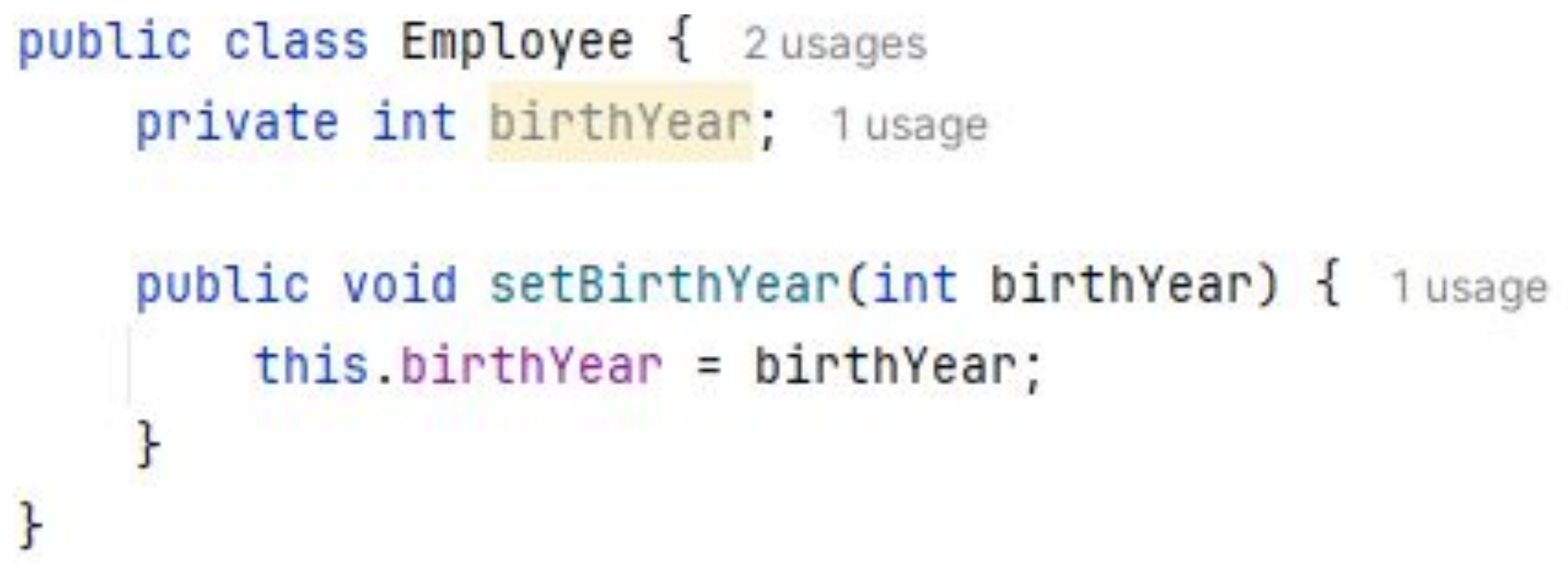
```java
public int balance; // Bad practice, no encapsulation
```

2. Improper Validation: Always validate data in setter methods to ensure the integrity of the object.

BAD EXAMPLE (Not Wrong)

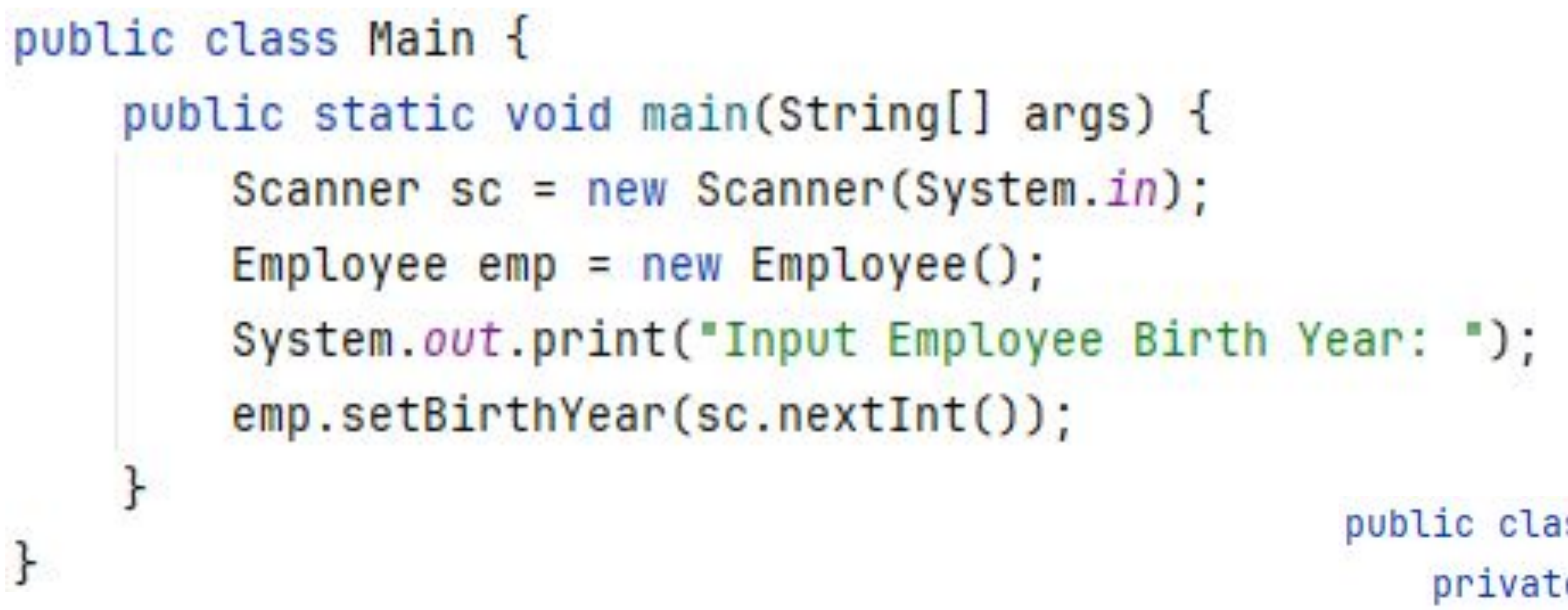


```java
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Employee emp = new Employee();
        System.out.print("Input Employee Birth Year: ");
        int birthYear = sc.nextInt();
        if( birthYear < Year.now().getValue() && birthYear > Year.now().getValue() - 100){
            emp.setBirthYear(birthYear);
        }
    }
}
```

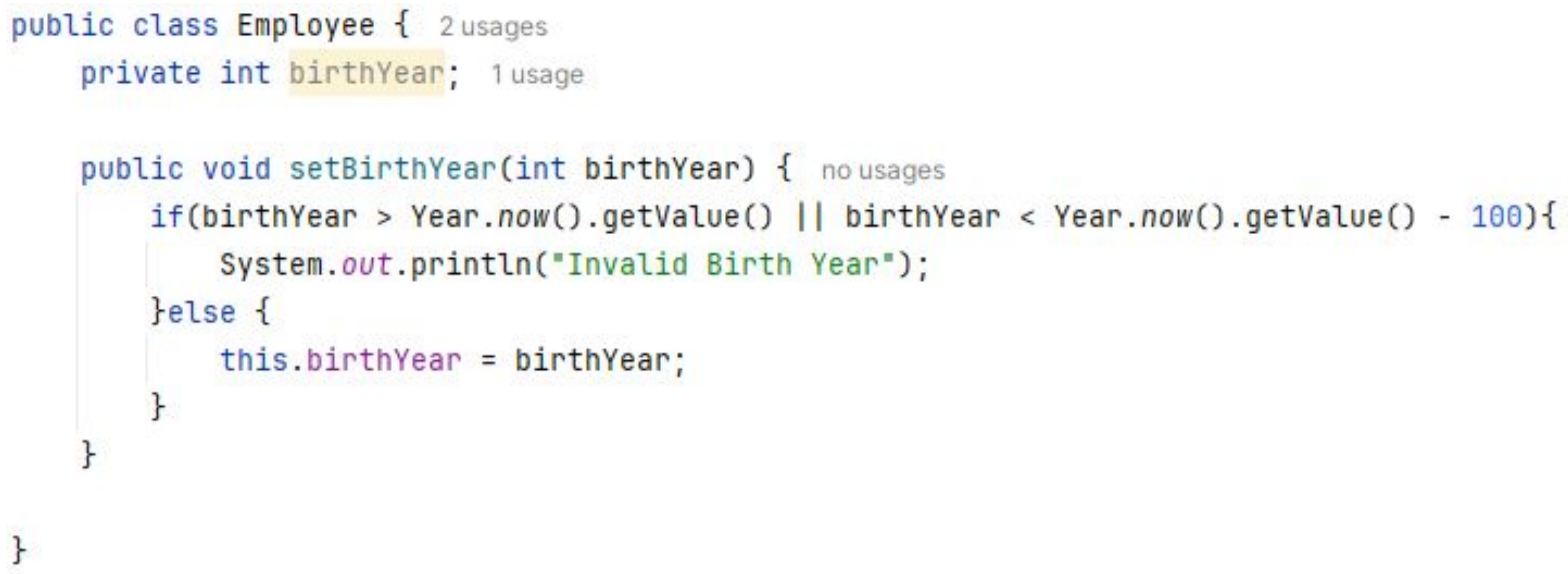


```java
public class Employee {  2 usages
    private int birthYear;  1 usage

    public void setBirthYear(int birthYear) {  1 usage
        this.birthYear = birthYear;
    }
}
```

BEST PRACTICE



```java
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Employee emp = new Employee();
        System.out.print("Input Employee Birth Year: ");
        emp.setBirthYear(sc.nextInt());
    }
}
```



```java
public class Employee {  2 usages
    private int birthYear;  1 usage

    public void setBirthYear(int birthYear) {  no usages
        if(birthYear > Year.now().getValue() || birthYear < Year.now().getValue() - 100){
            System.out.println("Invalid Birth Year");
        }else {
            this.birthYear = birthYear;
        }
    }

}
```

Real Case Scenario (Simplified)

Scenario: Bank Account Management

Problem: A bank needs to securely manage customer accounts. Each account has:

- accountNumber
- accountHolderName
- balance (modifiable with restrictions)

Encapsulation Implementation:

- Use private fields to protect sensitive data.
- Provide public methods to access or modify the data with validation.

Immutability & Read-Only

Immutability

Immutability means that once an object is created, its state (i.e., the values of its fields) cannot be changed.

Immutable objects are particularly useful in scenarios where you want to ensure the integrity of data, avoid accidental modifications, or work with threads safely in concurrent programming.

Characteristic of Immutable Class

1. All fields are private and final:
   - private ensures fields cannot be accessed directly.
   - final ensures that fields cannot be reassigned after they are initialized.
2. No setter methods:
   - Setter methods allow modification of fields, which must be avoided for immutability.
3. Fields are initialized through constructors:
   - Values of fields are set during object creation and cannot be changed afterward.
4. The class is declared final:
   - Declaring the class as final prevents it from being extended and its behavior from being overridden.
5. Mutable fields are avoided:
   - If the class contains mutable fields (like lists or objects), ensure they are not exposed directly and are safely handled.

BAN-PT
AKREDITASI INSTITUSI
UNGGUL

Thank You