Getters and Setters:

1. What Are Getters and Setters?

Getters and setters are methods used to **control access** to class variables.

- **Getter** → reads a value
- **Setter** → modifies a value

They enforce **encapsulation**, one of the core principles of Object-Oriented Programming (OOP).

Encapsulation means:

- Data is kept **private**
- Access is given through **controlled methods**

2. Why We Need Them (Real Reason)

If variables are public:

- Anyone can change them
- Invalid values can enter your object
- Debugging becomes painful

Getters and setters:

- Protect object state
- Allow validation
- Enable future changes without breaking code

3. Basic Example

```
class Student {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Key observations:

- Field is `private`
- Access is via `public` methods
- Internal data is hidden

---

4. Why Fields Should Always Be Private

Bad design:

```
student.name = "Abhinav";
```

Problems:

- No validation
- No rules
- No control

Correct design:

```
student.setName("Abhinav");
```

Now you control:

- What values are allowed
- How changes happen

---

5. Setter Validation (This Is Critical)

A setter should **protect the object**, not blindly assign values.

Bad setter:

```
public void setAge(int age) {
    this.age = age;
}
```

Good setter:

```
public void setAge(int age) {
    if (age <= 0) {
        throw new IllegalArgumentException("Age must be positive");
    }
    this.age = age;
}
```

Now your object cannot enter an invalid state.

---

6. Getters Can Do More Than Return Values

Getters can:

- Format output
- Compute values
- Hide internal representation

Example:

```
public String getFullName() {
    return firstName + " " + lastName;
```

```
}
```

There is no `fullName` variable, yet the class exposes a full name.

---

7. Standard Naming Convention (JavaBeans)

Frameworks rely on this naming.

Variable	Getter	Setter
name	getName()	setName()
age	getAge()	setAge()
active	isActive()	setActive()

Breaking this convention breaks:

- Spring
- Hibernate
- Jackson

---

8. When You Should NOT Write a Setter

Not every field should be mutable.

Example:

```
private final String id;

public String getId() {
    return id;
}
```

No setter means:

- State cannot be corrupted
- Object is safer

Immutability increases reliability.

---

9. Prefer Meaningful Methods Over Raw Setters

Bad design:

```
setBalance(double balance);
```

Better design:

```
deposit(double amount);
withdraw(double amount);
```

Why?

- Business rules remain safe
- Object controls its behavior
- Code reads naturally

---

10. Common Beginner Mistakes

- Making fields public
- Adding setters for everything
- No validation inside setters
- Getters exposing internal mutable objects
- Treating getters/setters as boilerplate

These mistakes cause fragile systems.

---

11. Best Practices Summary

- Always keep fields `private`
- Validate data inside setters
- Avoid unnecessary setters
- Use getters to hide implementation details

- Follow naming conventions strictly

---

12. Why Professionals Care About This

Getters and setters enable:

- Logging
- Security checks
- Debug tracking
- Framework integration
- Safe refactoring

They are dull on purpose. Stability looks boring.

---

13. What to Learn Next

- Immutable classes
- Defensive copying in getters
- Lombok: pros and cons
- Getters/setters in Spring and Hibernate
