1. Basic Method Overriding

Scenario:

We need to create an `Animal` class with a method `makeSound()`, and a subclass `Dog` that overrides this method.

Implementation Steps:

1. Define a **base class** `Animal` with a `makeSound()` method that prints `"Animal sound"`.
2. Create a **subclass** `Dog` that overrides `makeSound()` to print `"Bark"`.
3. Create objects of both classes and call the method to see overriding in action.

Expected Output:

```
Animal sound
Bark
```

---

2. Overriding with `@Override` Annotation

Scenario:

We need to override a method in a subclass and use the `@Override` annotation to ensure correctness.

Implementation Steps:

1. Define a `Vehicle` class with a `start()` method that prints `"Vehicle started"`.
2. Create a `Car` class that extends `Vehicle` and **overrides** `start()` with `@Override`.
3. Call `start()` from both `Vehicle` and `Car` objects to see the overridden method.

Expected Output:

```
Vehicle started
Car started
```

Why `@Override`?

- Ensures method signature is **exactly** the same as in the parent class.
- Helps avoid mistakes (e.g., typo in method name or incorrect parameters).

---

3. Overriding with Access Modifiers

Scenario:

A subclass can override a **protected** method from its parent and make it **more accessible**.

Implementation Steps:

1. Create a `Person` class with a **protected** method `display()` that prints `"I am a person"`.
2. Create a `Student` class that **overrides** `display()` as `public` and prints `"I am a student"`.
3. Call `display()` from a `Student` object.

Expected Output:

```
I am a student
```

Key Rule in Java:

- **Access level can be increased but not decreased** when overriding.
  - ▯ `protected` → `public` (Allowed)
  - ▯ `public` → `protected/private` (Not Allowed)

---

4. Method Overriding vs. Method Hiding (`static` methods)

Scenario:

We compare **overriding (instance methods)** with **hiding (static methods)** in Java.

Implementation Steps:

1. Define a `Parent` class with a **static** method `print()` that prints `"Parent"`.
2. Create a `Child` class that **also has** a static `print()` method printing `"Child"`.
3. Call `print()` using `Parent` and `Child` references.

Expected Output:

```
Parent
Child
```

Explanation:

- **Static methods are hidden, not overridden.**
- Method **binding happens at compile time**, not runtime.

---

5. Dynamic Method Dispatch (Runtime Polymorphism)

Scenario:

We demonstrate **runtime polymorphism**, where method calls are resolved at runtime.

Implementation Steps:

1. Create a `Shape` class with a `draw()` method printing `"Drawing shape"`.
2. Create `Circle` and `Square` subclasses that **override** `draw()`.
3. Store `Circle` and `Square` objects in a `Shape[]` array and call `draw()`.

Expected Output:

```
Drawing Circle
Drawing Square
```