🧠 Bridge Design Pattern – Simple Definition

Bridge pattern decouples an abstraction from its implementation so that both can **vary independently**.

🎯 Real-Life Analogy

Imagine a **Remote Control** and **TV**:

- You can have **multiple remotes** (BasicRemote, AdvancedRemote).
- You can have **multiple TVs** (SonyTV, LGTV).

You want to:

- Use **any remote with any TV**
- Add new remotes and TVs **without changing existing code**

✅ So, instead of tying Remote ↔ TV via inheritance, **we bridge them via composition.**

🔀 Why Not Inheritance?

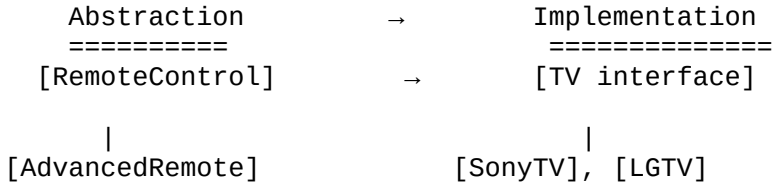
```
class SonyRemote extends SonyTV { }
class LGRemote extends LGTV { }
```

😵 Problem:

- If you add a new remote (`VoiceRemote`), you now need `SonyVoiceRemote`, `LGVoiceRemote`, etc.
- This causes **class explosion**.

✅ **Bridge Pattern** solves this!

🔧 Bridge Pattern Structure



🌉 Bridge Pattern – Java Example

✅ Step 1: Implementation Layer (TVs)

```
interface TV {
    void on();
    void off();
    void setChannel(int channel);
}

class SonyTV implements TV {
    public void on() { System.out.println("Sony TV turned ON"); }
    public void off() { System.out.println("Sony TV turned OFF"); }
    public void setChannel(int channel) {
        System.out.println("Sony TV: channel set to " + channel);
    }
}

class LGTV implements TV {
    public void on() { System.out.println("LG TV turned ON"); }
    public void off() { System.out.println("LG TV turned OFF"); }
    public void setChannel(int channel) {
        System.out.println("LG TV: channel set to " + channel);
    }
}
```

✅ Step 2: Abstraction Layer (Remote Controls)

```
abstract class RemoteControl {
    protected TV tv;

    public RemoteControl(TV tv) {
        this.tv = tv;
    }

    public void turnOn() { tv.on(); }
    public void turnOff() { tv.off(); }
    public abstract void setChannel(int channel);
}
```

✅ Step 3: Extended Abstraction

```
class BasicRemote extends RemoteControl {
    public BasicRemote(TV tv) {
        super(tv);
    }

    public void setChannel(int channel) {
        tv.setChannel(channel);
    }
}

class AdvancedRemote extends RemoteControl {
    public AdvancedRemote(TV tv) {
        super(tv);
    }
```

```
    public void setChannel(int channel) {
        System.out.println("Advanced remote: setting channel with fade
effect...");
        tv.setChannel(channel);
    }

    public void mute() {
        System.out.println("Muting the TV");
    }
}
```

✅ Step 4: Client Code

```
public class BridgePatternDemo {
    public static void main(String[] args) {
        TV sony = new SonyTV();
        RemoteControl remote1 = new BasicRemote(sony);
        remote1.turnOn();
        remote1.setChannel(10);

        TV lg = new LGTV();
        RemoteControl remote2 = new AdvancedRemote(lg);
        remote2.turnOn();
        remote2.setChannel(45);
        ((AdvancedRemote)remote2).mute();
    }
}
```

✅ Output:

```
Sony TV turned ON
Sony TV: channel set to 10
LG TV turned ON
Advanced remote: setting channel with fade effect...
LG TV: channel set to 45
Muting the TV
```

✅ Key Benefits of Bridge Pattern

Feature	Benefit
Separation of concerns	TV and Remote can evolve separately
Eliminates class explosion	No need for LGVoiceRemote, SonyVoiceRemote
Extensible	Add new TV or Remote without changing core
Runtime flexibility	Inject implementation dynamically

🆚 Bridge vs Adapter – Interview Tip

Adapter	Bridge
Connects **incompatible** interfaces	Separates **abstraction from implementation**
Used to **make old code work**	Used to **design new systems** cleanly

Adapter	Bridge
Focused on **conversion**	Focused on **extension and flexibility**

✅ Interview Answer

"The Bridge pattern decouples an abstraction (like Remote) from its implementation (like TV), so both can evolve independently. It uses **composition** to link them at runtime. It's particularly useful when you want to avoid a class explosion caused by combining multiple variants of two different dimensions. A real-world analogy is using any kind of remote with any TV without tight coupling."

Separation of concerns TV and Remote can evolve separately prove

🎯 What does "evolve separately" mean?

In plain terms:

- We can add **new TVs** without changing the Remote classes.
- We can add **new Remotes** without changing the TV classes.

This is the **separation of concerns** the Bridge pattern provides.

✅ Proof Step-by-Step (via Extension)

🔹 Step 1: Add a new TV (e.g., `SamsungTV`)

🚫 Without touching `RemoteControl`, `BasicRemote`, or `AdvancedRemote`.

```
class SamsungTV implements TV {
    public void on() { System.out.println("Samsung TV ON"); }
    public void off() { System.out.println("Samsung TV OFF"); }
    public void setChannel(int channel) {
        System.out.println("Samsung TV: channel = " + channel);
    }
}
```

✅ Use existing remote with new TV:

```
TV samsung = new SamsungTV();
RemoteControl remote = new BasicRemote(samsung);
remote.turnOn();             // Samsung TV ON
remote.setChannel(99);       // Samsung TV: channel = 99
```

✅ Remote works without modification = **TV evolved independently**

🔹 Step 2: Add a new Remote (e.g., `VoiceRemote`)

🚫 Without touching **any TV implementation**

```
class VoiceRemote extends RemoteControl {
    public VoiceRemote(TV tv) {
        super(tv);
    }

    public void setChannel(int channel) {
        System.out.println("Voice command: go to channel " + channel);
        tv.setChannel(channel);
    }

    public void say(String command) {
        System.out.println("Voice command received: " + command);
    }
}
```

✅ Use it with existing TV:

```
TV lg = new LGTV();
VoiceRemote voiceRemote = new VoiceRemote(lg);
voiceRemote.turnOn();            // LG TV turned ON
voiceRemote.setChannel(88);      // Voice command... LG TV: channel set to 88
voiceRemote.say("Mute");         // Voice command received: Mute
```

✅ TV works without modification = **Remote evolved independently**

🧠 Conclusion (Proof)

We were able to:

- Add **new TVs** without modifying or recompiling Remotes
- Add **new Remotes** without changing or recompiling any TV classes

✅ This proves **Bridge Pattern** allows **separation of concerns** and **independent evolution**, because **TV and Remote live in separate hierarchies** and communicate via an abstraction.

Runtime flexibility Inject implementation dynamically prove

Let’s now **prove at runtime** that you can switch between different implementations **without recompilation**, which is what "runtime flexibility" and "dynamic injection" mean.

✅ Goal

Can I inject a different `TV` implementation into the `RemoteControl` at **runtime**, without modifying or recompiling the remote?

✅ Reuse the Bridge Code:

```
interface TV {
    void on();
    void off();
    void setChannel(int channel);
}

abstract class RemoteControl {
    protected TV tv;

    public RemoteControl(TV tv) {
        this.tv = tv;
    }

    public void turnOn() { tv.on(); }
    public void turnOff() { tv.off(); }
    public abstract void setChannel(int channel);
}
```

🔁 Proof: Dynamic Selection at Runtime

```
public class BridgeRuntimeTest {
    public static void main(String[] args) {
        String tvType = "sony"; // Simulate runtime input, e.g., from config

        TV tv = getTVFromConfig(tvType); // INJECT AT RUNTIME
        RemoteControl remote = new BasicRemote(tv);

        remote.turnOn();
        remote.setChannel(5);
    }

    public static TV getTVFromConfig(String tvType) {
        return switch (tvType.toLowerCase()) {
            case "sony" -> new SonyTV();
            case "lg"   -> new LGTV();
            case "samsung" -> new SamsungTV();
            default -> throw new IllegalArgumentException("Unsupported TV");
        };
    }
}
```

✅ Output (when `tvType = "lg"`):

```
LG TV turned ON
LG TV: channel set to 5
```

Now change the config:

```
java
CopyEdit
String tvType = "samsung";
```

Output:

```
graphql
CopyEdit
Samsung TV ON
Samsung TV: channel = 5
```

✅ What Just Happened?

- The `RemoteControl` **abstraction stayed the same**
- The **TV implementation changed dynamically**
- No `if-else` in `RemoteControl`, no recompilation
- Behavior is dynamically injected → **This is runtime flexibility**

✅ Final Interview One-Liner

> "Bridge provides runtime flexibility by letting us inject implementations (like `TV`) into abstractions (like `RemoteControl`) at runtime using composition. This allows systems to be extended or configured without recompilation — one of the key benefits over static inheritance."

Would you like to connect this to **Spring's dependency injection**, where beans are injected at runtime (another real-world proof of Bridge concept)?

You said:
Would you like to connect this to **Spring's dependency injection**, where beans are injected at runtime (another real-world proof of Bridge concept)?

Perfect! Let's now connect the **Bridge Pattern** to **Spring Boot's Dependency Injection** with a **realistic example** that proves:

✅ Abstractions and implementations are decoupled
✅ Implementations are injected at runtime
✅ New implementations can be added **without changing** the abstraction

---

✅ Scenario: Controlling TVs via Remote — Spring Boot Style

Let's simulate a **Bridge** pattern in Spring Boot:

- `TV` = Implementation hierarchy
- `RemoteControl` = Abstraction hierarchy

- Spring Boot will **inject the TV implementation at runtime** into the `RemoteControl`

🧱 1. Define TV Interface and Implementations

```
public interface TV {
    void on();
    void off();
    void setChannel(int channel);
}


@Component("sonyTV")
public class SonyTV implements TV {
    public void on() { System.out.println("Sony TV ON"); }
    public void off() { System.out.println("Sony TV OFF"); }
    public void setChannel(int channel) {
        System.out.println("Sony TV: Channel = " + channel);
    }
}

@Component("lgTV")
public class LGTV implements TV {
    public void on() { System.out.println("LG TV ON"); }
    public void off() { System.out.println("LG TV OFF"); }
    public void setChannel(int channel) {
        System.out.println("LG TV: Channel = " + channel);
    }
}
```

🎮 2. Define `RemoteControl` Abstraction

```
java
CopyEdit
public abstract class RemoteControl {
    protected TV tv;

    public RemoteControl(TV tv) {
        this.tv = tv;
    }

    public void turnOn() { tv.on(); }
    public void turnOff() { tv.off(); }
    public abstract void setChannel(int channel);
}
```

🕹️ 3. Concrete Remote Implementation

```
java
CopyEdit
@Component
public class BasicRemote extends RemoteControl {

    // Inject TV at runtime using constructor injection
    public BasicRemote(@Qualifier("lgTV") TV tv) {
        super(tv); // bridge via composition
```

```
    }

    @Override
    public void setChannel(int channel) {
        System.out.println("Remote: Changing channel...");
        tv.setChannel(channel);
    }
}
```

You can swap `"lgTV"` with `"sonyTV"` in the `@Qualifier` **without touching** `RemoteControl` or `TV`.

🚀 4. Spring Boot Application (Client Code)

```
java
CopyEdit
@SpringBootApplication
public class BridgePatternSpringApp {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
SpringApplication.run(BridgePatternSpringApp.class, args);
        RemoteControl remote = context.getBean(BasicRemote.class);

        remote.turnOn();
        remote.setChannel(101);
        remote.turnOff();
    }
}
```

✅ Output (if `@Qualifier("lgTV")`):

```
vbnet
CopyEdit
LG TV ON
Remote: Changing channel...
LG TV: Channel = 101
LG TV OFF
```

🎯 Change Only This:

```
java
CopyEdit
// From:
public BasicRemote(@Qualifier("lgTV") TV tv)

// To:
public BasicRemote(@Qualifier("sonyTV") TV tv)
```

✅ Output switches to SonyTV — **no other class is touched.**

✅ Interview Summary

"Spring Boot proves the Bridge Pattern in practice by injecting implementations (like `SonyTV`) into abstractions (like `RemoteControl`) via composition. The implementation can vary at runtime via configuration (`@Qualifier`, `@Profile`, etc.), without altering the abstraction — the very definition of the Bridge pattern."

Would you like to take this further with:

- A `@Profile`-based dynamic TV selector?
- YAML-based injection (bridge via config)?
- Unit test for mocking TV implementations?

Let me know how deep you'd like to go.

real-world JDK or Spring use case that shows this in action (like `JDBC` or `Logger` abstraction)?

Let's connect the **Bridge Pattern** to real-world examples from both:

✅ 1. JDK Use Case (Bridge Pattern in JDBC)

✅ 2. Spring Boot Use Case (Logger, Messaging, etc.)

Then we'll show:

✅ 3. Bridge via `@Profile` in Spring

✅ 4. Bridge via `application.yml`

✅ 1. JDK – JDBC Is a Classic Bridge Pattern

🎯 Abstraction: `java.sql.Connection`, `Statement`, `ResultSet`

🛠️ Implementation: MySQL, Oracle, H2, PostgreSQL, etc.

```
java
CopyEdit
Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/test", "user", "password"
);
```

- `DriverManager` is the **bridge**, returning a `Connection` instance.
- Actual implementation (e.g., `com.mysql.jdbc.ConnectionImpl`) is injected **at runtime** by loading driver JAR.

- You don’t touch the abstraction, only change the driver or URL — classic **Bridge pattern** in JDK.

✅ 2. Spring Boot – Logging System as Bridge

🎯 Abstraction: `org.slf4j.Logger`

🛠️ Implementation: Logback, Log4j2, java.util.logging, etc.

```
java
CopyEdit
@Slf4j
@RestController
public class ProductController {

    @GetMapping("/test")
    public String test() {
        log.info("Inside test()");
        return "Working";
    }
}
```

- You always use `Logger` interface.
- Behind the scenes, the implementation (Logback, Log4j2, etc.) is injected **via** `spring-boot-starter-logging`.
- You can switch logging backend **without changing any business logic**.

✅ 3. Spring Bridge Pattern via `@Profile`

🔧 Use case: Inject different `TV` implementations based on environment

✅ Step 1: Define TVs with `@Profile`

```
java
CopyEdit
@Component
@Profile("sony")
public class SonyTV implements TV {
    public void on() { System.out.println("Sony TV ON"); }
    public void off() { System.out.println("Sony TV OFF"); }
    public void setChannel(int channel) {
        System.out.println("Sony TV: Channel = " + channel);
    }
}

@Component
@Profile("lg")
public class LGTV implements TV {
    public void on() { System.out.println("LG TV ON"); }
    public void off() { System.out.println("LG TV OFF"); }
    public void setChannel(int channel) {
        System.out.println("LG TV: Channel = " + channel);
```

```
    }
}
```

✅ Step 2: Inject TV into Remote (Bridge)

```
java
CopyEdit
@Component
public class VoiceRemote extends RemoteControl {
    public VoiceRemote(TV tv) {
        super(tv);
    }

    public void setChannel(int channel) {
        System.out.println("Voice: Go to channel " + channel);
        tv.setChannel(channel);
    }
}
```

✅ Step 3: Activate Profile in `application.yml`

```
yaml
CopyEdit
spring:
  profiles:
    active: lg
```

> 🔁 Switch `lg` to `sony` and **without changing Java code**, different implementation is injected.

✅ 4. Bridge via YAML Property (Without @Profile)

Use YAML value to decide implementation **at runtime**, programmatically.

✅ Step 1: application.yml

```
yaml
CopyEdit
tv:
  brand: sony
```

✅ Step 2: Config + Factory

```
java
CopyEdit
@Configuration
public class TVFactory {

    @Value("${tv.brand}")
```

```
    private String brand;

    @Bean
    public TV tv() {
        return switch (brand.toLowerCase()) {
            case "sony" -> new SonyTV();
            case "lg" -> new LGTV();
            default -> throw new IllegalArgumentException("Invalid TV brand");
        };
    }
}
```

✅ Step 3: Use TV in Remote

```
java
CopyEdit
@Component
public class BasicRemote extends RemoteControl {

    public BasicRemote(TV tv) {
        super(tv);
    }

    public void setChannel(int channel) {
        tv.setChannel(channel);
    }
}
```

✅ Now, switching from `sony` to `lg` in YAML **changes behavior** with **zero code change** — that's **runtime bridging** using configuration.

✅ TL;DR: Interview-Ready Summary

> "Spring Boot and JDK use the Bridge pattern extensively. JDBC bridges `Connection` abstraction with vendor-specific drivers. Spring bridges `Logger`, `DataSource`, and even business interfaces using composition and DI. Through `@Profile` and YAML, Spring allows injecting implementation at runtime — proving the power of the Bridge pattern."

What is the meaning of ->

Great observation! That diagram you saw:

```
Abstraction             →     Implementation
==========                    ==============
[RemoteControl]         →       [TV interface]
     |                                |
[AdvancedRemote]                [SonyTV], [LGTV]
```

…was a **visual representation** of how the **Bridge Design Pattern** separates abstraction and implementation.

Let me now **explain it line-by-line** for clarity.

🔍 Meaning of the Diagram

1️⃣ `Abstraction → Implementation`

- The arrow (→) represents a **"has-a" relationship** or **delegation via composition**.
- This is the *bridge* itself — the core idea of the pattern.
- In other words:
  - `Abstraction` **uses** `Implementation` at runtime
  - NOT via inheritance, but by holding a reference to it.

2️⃣ `[RemoteControl] → [TV interface]`

- `RemoteControl` (abstract class) has a `TV` reference.
- `TV` is an **interface**, so different TVs (Sony, LG, etc.) can be injected.
- This arrow again means: RemoteControl **delegates** TV functionality to whatever `TV` object it's holding.

3️⃣ `[AdvancedRemote]` and `[SonyTV], [LGTV]`

- These are **concrete classes**:
  - `AdvancedRemote` is a **concrete subclass** of `RemoteControl`
  - `SonyTV`, `LGTV` are **implementations** of the `TV` interface

So this final part shows:

- On the left: a refined abstraction (`AdvancedRemote`)
- On the right: different concrete implementations (`SonyTV`, `LGTV`)

✅ TL;DR – What does → mean?

The arrow (→) symbolizes **composition**: the abstraction (like `RemoteControl`) **has a reference to** the implementation (like `TV`).
This is how the Bridge pattern works — instead of extending implementations, it **bridges** to them dynamically.
