

♣ Java Class Loading & Initialization Hierarchy

1 Passive Use (does not trigger class initialization)

Passive use can be further divided into:

A) No Loading at All

The class is **not even loaded** into JVM when you do these.

- Only the compiler checks that the class exists.
- JVM doesn't touch the class metadata yet.

Examples:

```
// Just a reference declaration (no loading)
MyClass ref;

// Declaring array (type known, array class created)
MyClass[] arr = null;

// instanceof check
boolean b = (obj instanceof MyClass);
```

☞ In all cases above, `MyClass` is not loaded/initialized yet.

B) Loaded, but Not Initialized

The JVM loads class metadata into Method Area (so Class object exists), but does **not** run static initializers or assign explicit static values.

Examples:

```
// Class literal
Class<?> c = MyClass.class;

// Accessing superclass or interfaces
Class<?> sup = MyClass.class.getSuperclass();
Class<?>[] inf = MyClass.class.getInterfaces();

// Accessing compile-time constants
System.out.println(ConstClass.VALUE); // static final int VALUE = 42;
```

☞ Here, `MyClass` is loaded, but **not initialized** until first *active use*.

2 Active Use (triggers class initialization)

Active use means JVM will **run static initializers, assign static values, and execute static blocks** immediately.

Examples:

```
// Creating object
MyClass obj = new MyClass();

// Accessing non-final static fields
System.out.println(MyClass.counter);

// Calling static methods
MyClass.doSomething();

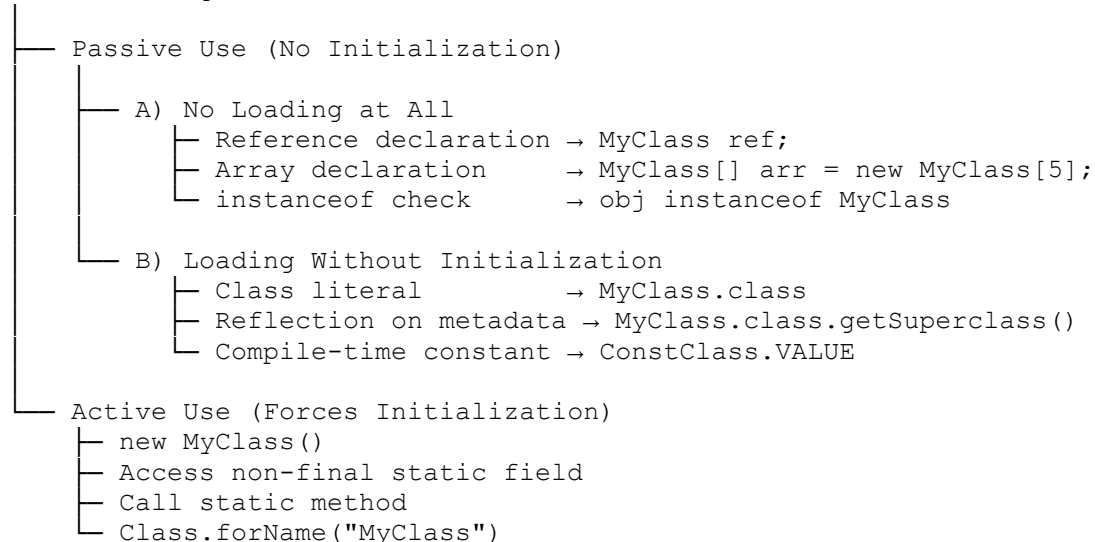
// Using Class.forName()
Class<?> c = Class.forName("MyClass");
```

☞ In all these cases:

- Class is loaded (if not already).
- Class is **linked** (verification, preparation, resolution).
- Class is **initialized** (static variables assigned, static blocks executed).

◆ Full Hierarchy (Visual Form)

Class Loading & Initialization



◆ Example Demonstration

```

class Test {
    static final int CONST = 10; // compile-time constant
    static int counter = 0;

    static {
        System.out.println("Test class initialized!");
    }

    public static void doSomething() {
        System.out.println("Static method called");
    }
}

public class Demo {
    public static void main(String[] args) throws Exception {
        Test t; // No loading
        Test[] arr = new Test[5]; // No loading
        boolean b = (null instanceof Test); // No loading

        Class<?> c1 = Test.class; // Loaded, not initialized
        System.out.println(Test.CONST); // Loaded, not initialized

        Test.doSomething(); // Active: initializes Test
        Class<?> c2 = Class.forName("Test"); // Active: initializes Test
    }
}

```

Output:

```

10
Test class initialized!
Static method called

```

Notice:

- "Test class initialized!" only appears when **active use** happens.
- Accessing `CONST` didn't initialize the class (because it's a compile-time constant).

✓ Summary

- **Passive (No Load):** reference, array, `instanceof`
- **Passive (Load but No Init):** class literal, reflection, constants
- **Active (Load + Init):** new instance, static field/method, `Class.forName()`

Do you want me, Engineer Mohamed, to also add **the JVM timeline (Loading → Linking → Initialization)** into this hierarchy, so you see exactly which phase happens in each case?

translate

Bilingual

Translation only

Hide translation

أنت قلت:

ok

قال ChatGPT:

Perfect 🦋 Engineer Mohamed — let's now enrich the hierarchy by adding the **JVM timeline** (Loading → Linking → Initialization) for each case, so you can clearly see *when each phase happens*.

♣ Java Class Loading Timeline

◆ JVM Phases (Quick Reminder)

1. **Loading** → Bytecode of class is read by the class loader into Method Area.
2. **Linking** (3 sub-steps):
 - *Verification* → bytecode validity check.
 - *Preparation* → memory for static variables, default values assigned.
 - *Resolution* → symbolic references → real references.
3. **Initialization** → Explicit static values assigned + static blocks executed.

⚡ Full Hierarchy with Timeline

① Passive Use

A) No Loading at All

The class is never even loaded into the JVM.

- Only the **compiler** checks that class exists.
- JVM doesn't perform *loading, linking, or initialization* yet.

Cases:

```
MyClass ref;           // Reference declaration
MyClass[] arr = null;   // Array declaration
boolean b = obj instanceof MyClass; // instanceof check
```

✓ Timeline:

- **Loading:** ✗ No
- **Linking:** ✗ No
- **Initialization:** ✗ No

B) Loading Without Initialization

The class is **loaded** + **linked** but **not initialized**.
Static blocks are not executed yet.

Cases:

```
Class<?> c = MyClass.class;           // Class literal
Class<?> sup = MyClass.class.getSuperclass(); // Reflection
System.out.println(ConstClass.VALUE); // Compile-time constant
```

✓ Timeline:

- **Loading:** ✓ Yes
- **Linking:** ✓ Yes (verification, preparation, resolution)
- **Initialization:** ✗ No

2 Active Use

Here, the class is **loaded** + **linked** + **initialized** immediately.
Static blocks run, static fields assigned.

Cases:

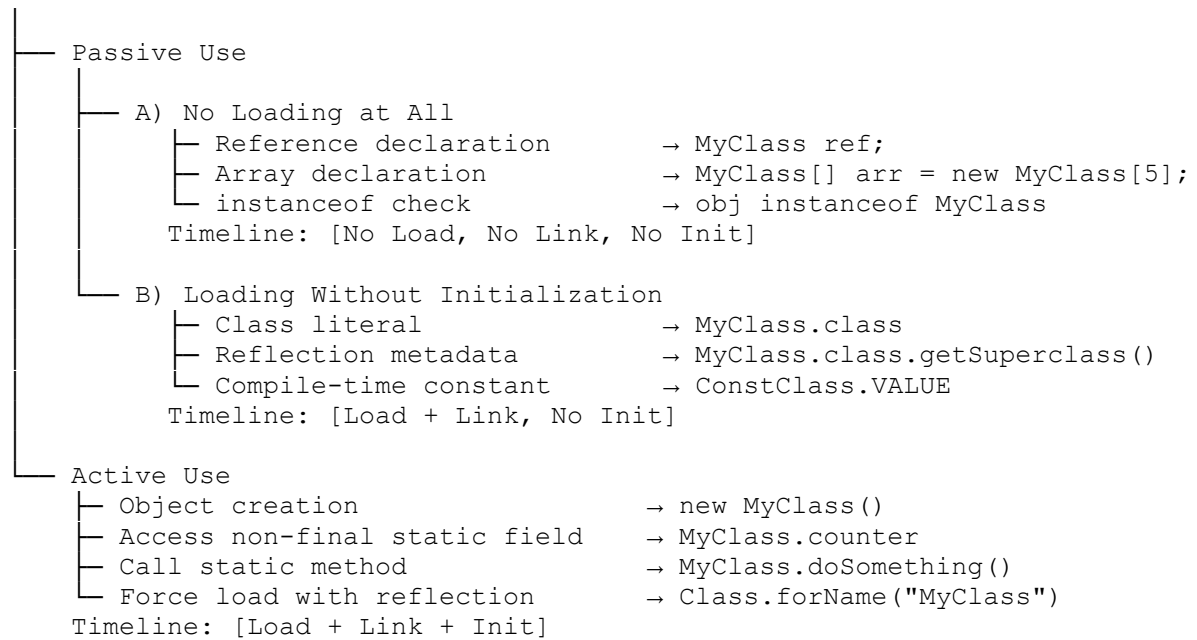
```
new MyClass();           // Instance creation
System.out.println(MyClass.counter); // Non-final static field
MyClass.doSomething();    // Static method
Class.forName("MyClass"); // Reflection active load
```

✓ Timeline:

- **Loading:** ✓ Yes
- **Linking:** ✓ Yes
- **Initialization:** ✓ Yes (static fields + static blocks executed)

Final Tree View (with JVM Phases)

Class Loading & Initialization



✓ **Key Rule to Memorize:**

- **Passive (A)** → Class is not touched at runtime.
- **Passive (B)** → Class is loaded + linked but not initialized.
- **Active** → Full loading + linking + initialization.