

Static vs Dynamic Binding



give me the deep and detailed explaination of Static vs Dynamic Binding in java so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explaination

Here is **the deepest, most detailed, and simplest explanation of Static Binding vs Dynamic Binding in Java**, with examples, diagrams, JVM-level understanding, and how it relates to **polymorphism, method overriding, method overloading, variables, references, and memory**.

★ STATIC VS DYNAMIC BINDING IN JAVA — COMPLETE DEEP EXPLANATION

Java uses the term **binding** to describe when the **link between a method call (or variable) and its actual code/definition is decided**.

There are two types:

1. **Static Binding** → Happens at **compile time**
2. **Dynamic Binding** → Happens at **runtime**

Let's explain both slowly and deeply.

◆ 1. What is BINDING?

Binding = Process of **linking a method call** to the **actual method body** that should run.

Example:

```
java
obj.show();
```

Here the compiler/JVM must decide:

- ✓ Which **show()** method should run?
- ✓ Where is the method defined?
- ✓ How will JVM execute it?

This decision can happen:

- **Before program runs** → compile-time → **Static Binding**

- While program runs → runtime → **Dynamic Binding**
-

◆ 2. STATIC BINDING (EARLY BINDING / COMPILE-TIME BINDING)

Static Binding occurs when the **compiler** decides which method or variable to use.

✓ When does static binding happen?

Whenever the **method call can be determined by the compiler based on the reference type** (NOT object type).

This includes:

Category	Binding Type
Method Overloading	Static Binding
Static Methods	Static Binding
Final Methods	Static Binding
Private Methods	Static Binding
Variables (fields)	Static Binding

These are bound **early** because:

- Private → Not accessible in child class → compiler knows the method already
 - Static → belongs to class → NOT overridable
 - Final → cannot be overridden
 - Overloaded → compiler decides based on parameter types
 - Variables → not polymorphic → chosen using reference type
-

◆ Example of Static Binding

```
java

class Test {
    private void display() { System.out.println("Private Display"); }
    static void show() { System.out.println("Static Show"); }
    final void result() { System.out.println("Final Result"); }

    void test(int a) { System.out.println("int version"); }
    void test(double a) { System.out.println("double version"); }
}

public class Main {
    public static void main(String[] args) {
        Test t = new Test();
        t.display(); // private
        t.show(); // static
        t.result(); // final
        t.test(10); // overloaded
    }
}
```

Who decides?

→ Compiler

Why?

Because **none** of these methods depend on runtime object type.

🎯 Key Characteristics of Static Binding

- ✓ Happens at **compile time**
 - ✓ Faster execution
 - ✓ Based on **reference type**
 - ✓ Does NOT support runtime polymorphism
 - ✓ Used with:
 - method overloading
 - private, static, final methods
 - variables
-

◆ 3. DYNAMIC BINDING (LATE BINDING / RUNTIME BINDING)

Dynamic Binding occurs when the **JVM** decides which method to call at **runtime**, based on the **actual object type**, NOT the reference type.

✓ When does dynamic binding happen?

Whenever **method overriding** is involved.

Overriding = same method signature in parent and child.

Why is it runtime?

Because:

At compile time → you only know the **reference type**

At runtime → JVM knows the **actual object in memory**

◆ Example of Dynamic Binding

```
java

class Parent {
    void show() { System.out.println("Parent Show"); }
}

class Child extends Parent {
    @Override
    void show() { System.out.println("Child Show"); }
}
```

```
public class Main {
    public static void main(String[] args) {
        Parent p = new Child();
        p.show(); // Which show() ??? Parent or Child
    }
}
```

What happens?

- Compiler **sees reference type = Parent**
- But JVM **sees object type = Child**

So JVM decides at **runtime** to call:

👉 **Child's show()**

This is **runtime polymorphism**.

🔥 Why Dynamic Binding is Needed?

To achieve:

- ✓ **Runtime Polymorphism**
- ✓ **Method Overriding**
- ✓ **Loose Coupling**
- ✓ **Flexibility**

Without dynamic binding, overriding would not work.

🧠 JVM INTERNALS: HOW IT DECIDES?

- On runtime, JVM stores a **virtual method table (vtable)**.
- For every object, JVM looks up **which overridden method** to call.
- Thus object type decides final method.

📌 STATIC VS DYNAMIC BINDING — VISUAL EXAMPLE

Example:

```
java
Parent p = new Child();
```

Compiler view:

- ✓ p is Parent
- ✓ So it checks only methods visible in Parent
- ✗ It doesn't know which method will run

JVM view:

- ✓ Object created = Child
- ✓ So call Child's overridden method

⚡ VARIABLE BINDING (IMPORTANT POINT)

Variables are **always static binding** in Java.

Example:

```
java

class Parent { int x = 10; }
class Child extends Parent { int x = 20; }

Parent p = new Child();
System.out.println(p.x); // 10
```

Why?

Because variables are NOT polymorphic.

Compiler decides using **reference type**.

📘 STATIC VS DYNAMIC BINDING — SUMMARY TABLE

Feature	Static Binding	Dynamic Binding
Decided When	Compile Time	Runtime
Done By	Compiler	JVM
Based On	Reference Type	Object Type
Supports	Method Overloading	Method Overriding
Used For	Private, Static, Final Methods + Variables	Non-final, non-static overridden methods
Performance	Faster	Slightly slower
Polymorphism	No	Yes
Example	<code>t.test(10)</code>	<code>p.show()</code> (overridden)

🧩 FULL COMPARISON WITH CODE (IMPORTANT EXAMPLE)

```
java

class A {
    void run() { System.out.println("A running"); }
```

```

    static void display() { System.out.println("A static display"); }
}

class B extends A {
    void run() { System.out.println("B running"); }
    static void display() { System.out.println("B static display"); }
}

public class Test {
    public static void main(String[] args) {

        A obj = new B();

        obj.run();      // Dynamic Binding → B running
        obj.display(); // Static Binding → A static display
    }
}

```

Why different?

- `run()` → overridden → **dynamic binding**
- `display()` → static → **static binding** → depends on reference type `A`

🔥 THE MOST IMPORTANT RULES TO REMEMBER

- ✓ Rule 1: Overloading → **static binding**
- ✓ Rule 2: Overriding → **dynamic binding**
- ✓ Rule 3: **Static, Final, Private** → **static binding**
- ✓ Rule 4: **Variables** → **static binding**
- ✓ Rule 5: **Dynamic binding works only with overridable instance methods**

💡 WHY JAVA USES BOTH?

Static binding:

- For faster execution
- For methods that cannot be overridden
- For compile-time safety

Dynamic binding:

- For runtime polymorphism
- For flexible code architecture

🧠 REAL-LIFE ANALOGY

Think of **binding** like finalizing a delivery address:

Static Binding:

Address is printed on the package before shipping → cannot change later. (fixed at compile time)

Dynamic Binding:

Delivery boy asks at runtime "Where should I deliver?" → depends on actual location.
(overriding, runtime behavior)

FINAL REVISION (ONE-LINERS)

- **Static Binding** = compile-time = reference type = overloading + static/final/private
 - **Dynamic Binding** = runtime = object type = overriding
 - Only **instance overridable methods** use dynamic binding.
-

If you want, I can also provide:

- Mind map diagram
- Memory tricks
- 20 interview questions on static vs dynamic binding
- Advanced JVM internal working

Just tell me!