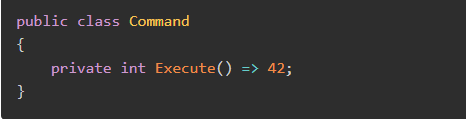
Got it! I’ll break it down step by step in the simplest way possible. Let’s go! 🚀



**Understanding the Problem: Why Optimize Reflection?**

* **Reflection** allows us to inspect and call methods dynamically (even private ones).
* But **it’s slow** because it finds methods at runtime, rather than compile-time.
* So, if we use reflection in a **loop or frequently**, our code **gets sluggish**.

We’ll look at **four** different ways to improve this:  
1️ **Basic Reflection** (slowest)  
2️ **Cached Reflection** (better)  
3️ **Delegate-Based Approach** (way better)  
4️ **Compiled Expressions** (best)

**1️⃣ Basic Reflection (Slow)**

This is the simplest way to call a **private method** using reflection.

**🔹 Code:**

**🔹 How It Works:**

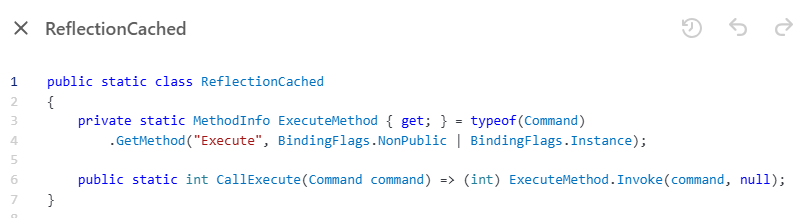
* typeof(Command): Gets the type of the Command class.
* .GetMethod("Execute", BindingFlags.NonPublic | BindingFlags.Instance): Finds the **private method** named "Execute".
* .Invoke(command, null): Calls that method on the command object.

✅ **Pros:** It works!  
❌ **Cons:** **Every time** we call it, it has to **find the method again**, which is slow!

**2️ Cached Reflection (A Bit Faster)**

Instead of searching for the method **every time**, let’s find it **once** and store it.

**🔹 Code:**

**🔹 What’s Different?**

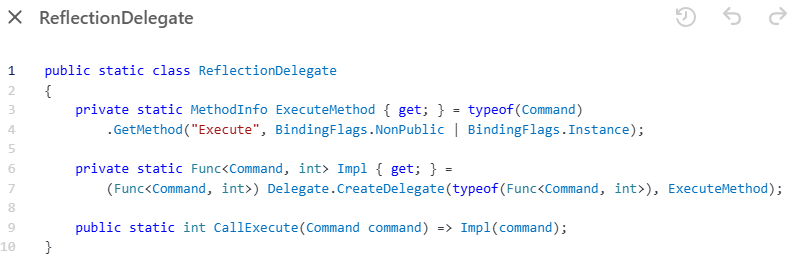
* We **store** the method (ExecuteMethod) **once**.
* Then, every time we need to call it, we **just use** ExecuteMethod.Invoke(command, null).

✅ **Pros:** **Faster** than basic reflection because we don’t search every time.  
❌ **Cons:** Invoke() is still **slow** because it uses late binding.

**3️ Delegate-Based Approach (Much Faster)**

Instead of using .Invoke(), we create a **delegate** (a function pointer).

**🔹 Code:**

**🔹 What’s Different?**

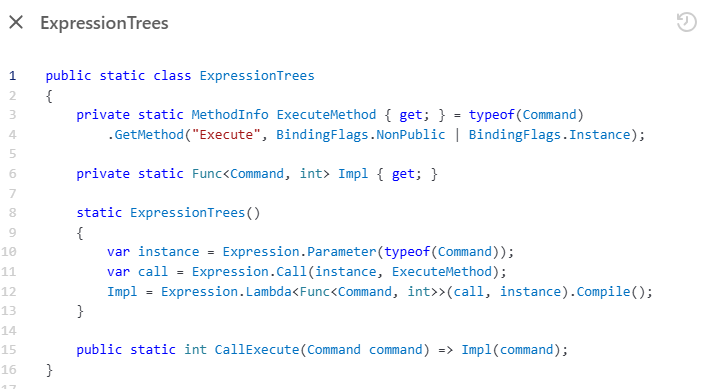
* **Delegate.CreateDelegate(...)**: Converts the method into a **function pointer**.
* **Instead of Invoke()**, we call Impl(command), which is **way faster**!

✅ **Pros:** Much **faster** than reflection!  
❌ **Cons:** Can’t be used for **constructors** or certain method types.

**4️ Compiled Expressions (Fastest!)**

This method **generates code at runtime**, so it's **as fast as normal method calls**.

**🔹 Code:**

**🔹 What’s Happening?**

1. **Find the method (ExecuteMethod)** like before.
2. **Create an Expression tree**, which represents the method call.
3. **Compile it** into real executable code.
4. **Now calling Impl(command) is just like calling a normal method!**

✅ **Pros:** 🚀 **Fastest method!**  
❌ **Cons:** A bit more complex.

**Performance Comparison (Benchmark Results)**

We test these methods using Benchmark.NET. Here’s how fast each method is:

| **Method** | **Time (ns)** | **Relative Speed** |
| --- | --- | --- |
| **Basic Reflection** | 192.975 ns | ❌ **(Slowest)** |
| **Cached Reflection** | 123.762 ns | ✅ **(~2x faster)** |
| **Delegate-Based Approach** | 6.419 ns | 🚀 **(~30x faster)** |
| **Compiled Expressions** | 5.383 ns | 🔥 **(Fastest!)** |

🚀 **Conclusion:** If you need to call private methods efficiently:

* Use **Compiled Expressions** for **maximum performance**.
* If that’s overkill, **Delegates** (CreateDelegate(...)) are almost as good.

Hope that made it easy! Let me know if anything needs more explanation. 😊

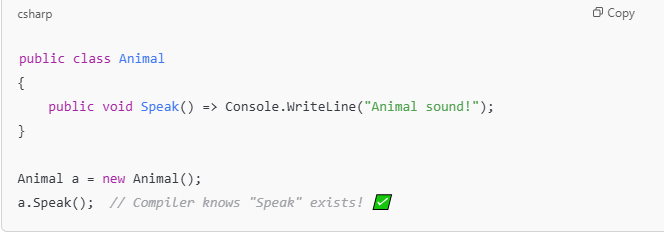
Great question! Let's break it down **super simply**.

# What is Late Binding? 🤔

🔹 **Late binding** means the method or property being called is determined **at runtime (while the program is running)** instead of **at compile-time (when the code is being compiled).**

**Example of Early Binding (Fast)**

🚀 **Early binding** means the compiler **already knows** which method will be called.  
This makes it **fast** because everything is checked **before** the program runs.

✅ **Advantages of Early Binding**:

* Faster execution
* Catches errors at compile-time

**Example of Late Binding (Slower)**

🚀 **Late binding** means the method is **found dynamically at runtime**, which is **slower**.  
It happens in **Reflection**, **dynamic types**, or **interfaces without knowing the exact type**.

🔹 The compiler **does not** know if "Speak" exists.  
🔹 The program must **search for the method** while running.

✅ **Pros of Late Binding**:

* Can call methods dynamically (useful in plugins, frameworks).  
  ❌ **Cons**:
* **Slower** (because it must find the method at runtime).
* **Error-prone** (if the method name is wrong, it crashes at runtime).

**Where Do We Use Late Binding?**

* **Reflection** (MethodInfo.Invoke())
* **Dynamic Types** (dynamic keyword)
* **COM objects** in older Windows programming
* **Plugins & Scripting** (e.g., loading external DLLs dynamically)

Hope this makes sense! 🚀 Let me know if you want more examples. 😊

# How Does Cached Reflection Work? 🧐

Reflection is **slow** because every time we call GetMethod(), it **searches the class for the method** at runtime. **Caching** avoids that by storing the method info **once** and reusing it.

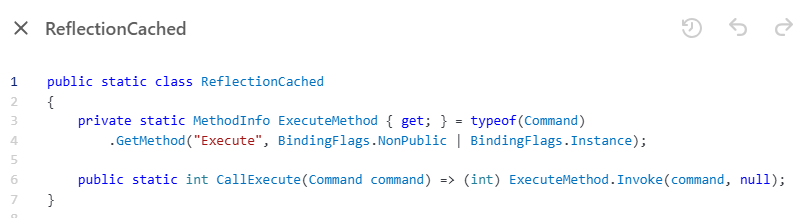
**🔴 Problem with Normal Reflection (Slow)**

**Why is it slow?**

* **GetMethod("Execute") searches the class every time** the function runs.
* **If called in a loop, it wastes time**.

**🟢 Solution: Cached Reflection (Faster)**

Instead of searching for the method every time, **we store it once in a static field** and use it when needed.



**⚙️ How Does It Work?**

1️ **When the class is first used**,

* ExecuteMethod is assigned **once** using GetMethod().
* This happens **only once** (when the class is first accessed).

2️ **Every time we call CallExecute()**,

* It uses the **pre-stored MethodInfo** instead of searching for it again.

**🛠️ Why Is Cached Reflection Faster?**

| **Approach** | **Performance** | **Why?** |
| --- | --- | --- |
| Normal Reflection | **Slow** 🐌 | GetMethod() searches for the method **every time** |
| Cached Reflection | **Faster** 🚀 | GetMethod() runs **only once**, then reuses the stored method |

By **reducing repeated method lookups**, cached reflection improves performance significantly. 🎯

# Why cached Reflection is faster than normal Reflection?

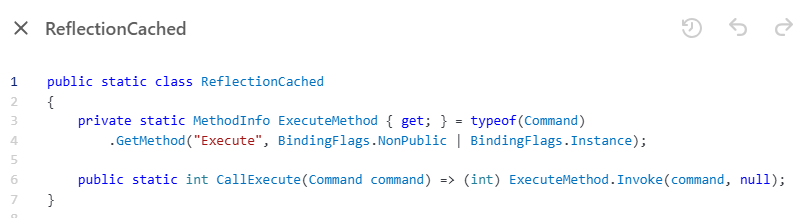
gGreat observation! The key difference is **where the GetMethod() call happens** and whether it is evaluated **once at class initialization** or **every time the method is called**. Let's break it down step by step.

**🔴 Normal Reflection (Inefficient)**

**⚡ Why is GetMethod() called multiple times?**

* The GetMethod() call happens **inside the method body** of CallExecute().
* Every time you call CallExecute(), the code **executes from the top**.
* So, GetMethod() runs **every single time** CallExecute() is called, making it inefficient.

**✅ Cached Reflection (Efficient)**

**⚡ Why is GetMethod() called only once?**

* ExecuteMethod is a **static field**, and **static fields are initialized once per class, when it is first used**.
* Since ExecuteMethod is **outside** CallExecute(), it is evaluated **only once** when the ReflectionCached class is first accessed.
* After that, CallExecute() just **reuses the stored method info** without calling GetMethod() again.

**🚀 Why Normal Reflection Doesn’t Store GetMethod() Once?**

Even though CallExecute() is a **static method**, the important part is that GetMethod() is **inside the method body**.

* Static methods **don’t store local variables between calls**—each time you call CallExecute(), it starts fresh.
* Since GetMethod() is inside the method body, it **executes again** every time the method runs.

📌 **Key Rule:**

* **Static fields** (like ExecuteMethod in ReflectionCached) are evaluated **only once**, when the class is first loaded.
* **Method-local variables** (like GetMethod() inside CallExecute()) are **re-evaluated every time the method is called**.

**⏳ Performance Comparison**

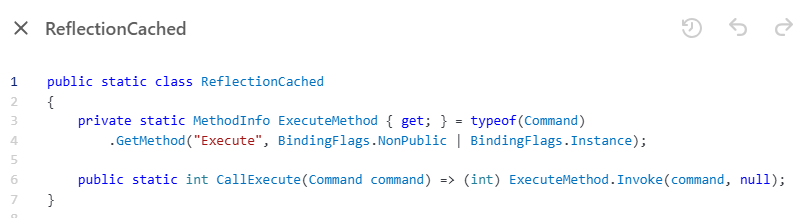
| **Approach** | **GetMethod() Location** | **Called Every Time?** | **Efficiency** |
| --- | --- | --- | --- |
| **Normal Reflection** | Inside CallExecute() | 🔴 Yes | ❌ Slow |
| **Cached Reflection** | Static Field | 🟢 No (Only Once) | ✅ Faster |

So, **even though both CallExecute() methods are static**, only the **cached approach stores MethodInfo in a static field**, which prevents GetMethod() from running every time.

# How Does the Delegate-Based Approach Work? 🏎️

A **delegate** is like a function pointer—it stores a reference to a method and allows us to call it **directly**, skipping the overhead of MethodInfo.Invoke().

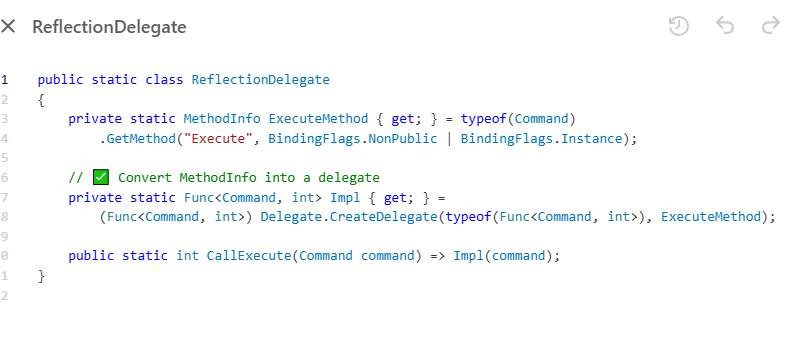
**🔴 Cached Reflection (Better, But Still Slow)**

✅ **Why is it better than normal reflection?**

* **Avoids repeated calls to GetMethod()** (caches the method info).  
  ❌ **But still slow because** MethodInfo.Invoke() uses **late binding**, adding overhead.

**🟢 Delegate-Based Approach (Even Faster!)**

Instead of using MethodInfo.Invoke(), we **convert the method into a delegate** that we can call directly.



**⚙️ How Does It Work?**

1️ **Find the method once** using GetMethod().  
2️ **Convert MethodInfo into a delegate** using Delegate.CreateDelegate().

* This allows **direct method calls** instead of Invoke().  
  3️⃣ **Call the method using the delegate**, which is much faster.

**🛠️ How Is This Different from Cached Reflection?**

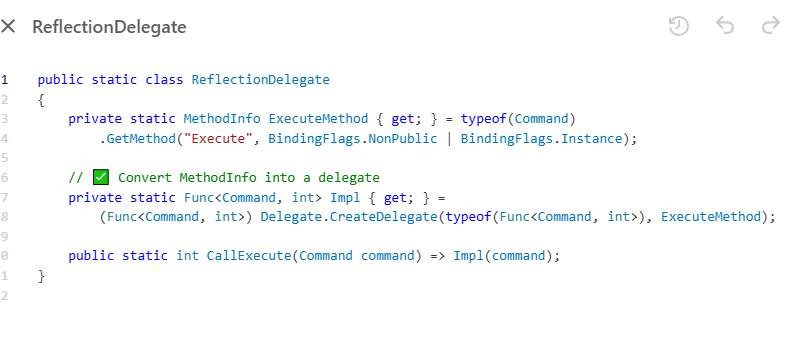
| **Approach** | **Performance 🚀** | **How It Works** |
| --- | --- | --- |
| **Normal Reflection** | **Slow 🐌** | Calls GetMethod() **every time**. |
| **Cached Reflection** | **Faster 🚀** | Stores MethodInfo, but still uses Invoke() (late binding). |
| **Delegate Approach** | **Fastest ⚡** | Converts MethodInfo into a delegate and calls the method **directly**. |

✅ **Why Is Delegation Faster?**

* **No Invoke() overhead** (direct method calls).
* **Avoids reflection overhead** after the initial setup.

Hope that makes sense! Let me know if you need further clarification. 🚀😊

# 🔍 How Does the Delegate Approach Work?



**⚡ How is this different from Cached Reflection?**

1. **Cached Reflection** still calls MethodInfo.Invoke(command, null);
   * This method uses **late binding**, meaning the runtime has to process the method call dynamically.
   * This adds overhead because it must handle type conversions, boxing/unboxing, and security checks every time.
2. **Delegate Approach** **directly calls** the function **without using Invoke()**.
   * Delegate.CreateDelegate(...) creates a **direct function pointer** to Execute().
   * This allows us to call it **like a normal function**, avoiding reflection overhead.

**⏳ How Does It Work Internally?**

1. **Step 1**: GetMethod() finds the private Execute() method.
2. **Step 2**: Delegate.CreateDelegate(...) **creates a direct function pointer** for Execute().
   * Now Impl(command) behaves exactly like calling command.Execute() directly.
3. **Step 3**: Calling CallExecute(command) simply **calls the delegate**, which is **much faster** than reflection.

**⏳ Performance Boost**

| **Approach** | **How it Calls Execute()** | **Overhead** | **Speed** |
| --- | --- | --- | --- |
| **Reflection** | MethodInfo.Invoke() | 🔴 High | ❌ Slow |
| **Cached Reflection** | MethodInfo.Invoke() (cached) | 🟡 Medium | ✅ Faster |
| **Delegate Approach** | **Direct Function Call** via delegate | 🟢 Low | 🚀 Fastest |

🔹 **Why is it fast?** Because MethodInfo.Invoke() is completely skipped, and the function is called **directly** via the delegate.

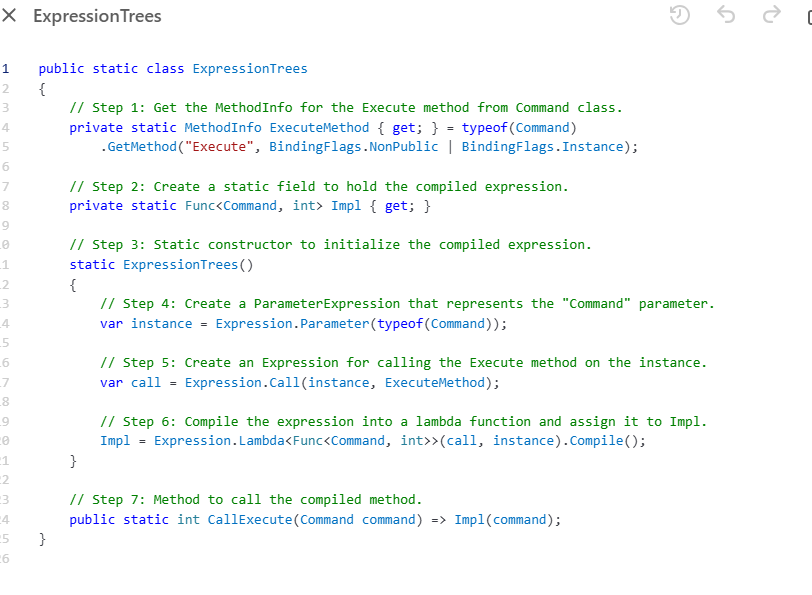
**📌 Key Takeaway**

* Instead of **looking up and calling** the method at runtime (**reflection**), the delegate approach **stores a direct reference** to the function.
* This makes it **as fast as calling a normal method**, without reflection overhead!

Hope this makes sense! 🚀🔥 Let me know if you need more details! 😎

Absolutely! Let's go step by step with the code example and then explain how everything works.

**Code Example for Compiled Approach (Expression Trees)**



**Explanation of Each Block (with Comments)**

1. **Get the MethodInfo for the Execute Method**

private static MethodInfo ExecuteMethod { get; } = typeof(Command)

.GetMethod("Execute", BindingFlags.NonPublic | BindingFlags.Instance);

* **What’s happening here?**
  + We are **finding the method** Execute in the Command class using reflection.
  + The method is **private**, so we use BindingFlags.NonPublic to access it, and BindingFlags.Instance specifies it's an instance method.
* **Why is it needed?**
  + This will allow us to **call** the Execute method dynamically later, using the expression tree.

1. **Create a Static Field to Hold the Compiled Expression**

private static Func<Command, int> Impl { get; }

* **What’s happening here?**
  + We declare a static field Impl, which is a delegate of type Func<Command, int>. This delegate will **hold the compiled expression** of calling the Execute method.
* **Why is it needed?**
  + This delegate will **store the compiled method call**, so it can be used multiple times without re-compiling.

1. **Static Constructor to Initialize the Compiled Expression**

static ExpressionTrees()

{

// Create expression here

}

* **What’s happening here?**
  + The **static constructor** is used to set up the compiled expression when the class is first accessed.
* **Why is it needed?**
  + The static constructor is executed only once, making the setup efficient. **It ensures that all heavy-lifting is done once and only once**.

1. **Create a ParameterExpression for the Command Instance**

var instance = Expression.Parameter(typeof(Command));

* **What’s happening here?**
  + We create a **parameter expression** for the Command object. This represents the **parameter** to the method we will call, i.e., the Command instance.
* **Why is it needed?**
  + The parameter represents the **input to the method** when we later call it dynamically. It's part of the expression tree that will be compiled.

1. **Create an Expression to Call the Execute Method**

var call = Expression.Call(instance, ExecuteMethod);

* **What’s happening here?**
  + We create an **expression** to **call** the Execute method on the Command instance.
  + Expression.Call is a method that lets us represent a **method invocation** in our expression tree.
* **Why is it needed?**
  + This step is crucial for **defining the actual logic** (calling the Execute method). It turns the method into an expression we can compile.

1. **Compile the Expression into a Lambda Function**

Impl = Expression.Lambda<Func<Command, int>>(call, instance).Compile();

* **What’s happening here?**
  + We turn the expression tree (which contains the method call) into a **lambda function** (Func<Command, int>). This lambda represents the compiled method call.
  + Finally, we **compile** the lambda into executable code and store it in the Impl delegate.
* **Why is it needed?**
  + **Compiling** turns the expression tree into fast, executable code. This **eliminates the need for reflection** on subsequent calls and speeds up execution.

1. **Call the Compiled Method**

public static int CallExecute(Command command) => Impl(command);

* **What’s happening here?**
  + This is the actual **method** that will be called. It simply invokes the **compiled function** stored in Impl.
* **Why is it needed?**
  + This **uses the pre-compiled method**, which is **faster** than reflection or other approaches because it doesn’t involve looking up the method every time.

**Key Points to Remember:**

* **Expression Tree**: We’re building an expression tree that represents the method call. Think of this as a **blueprint** for the dynamic method.
* **Compilation**: The blueprint is **compiled** into executable code, which is then used repeatedly without performance overhead.
* **Delegate**: The compiled expression is stored in a **delegate**, which acts like a fast, reusable function.

**Why This Approach is Efficient:**

* **One-Time Compilation**: The expression tree is compiled **only once** and is stored in a delegate for reuse, making future calls **very fast**.
* **No Reflection at Runtime**: There’s no slow reflection process every time we invoke the method. It’s just a **direct method call** after the initial compilation.
* **Optimized for Repeated Calls**: Once compiled, calling the method is nearly as fast as a regular C# method call, which **greatly improves performance** in tight loops or repeated calls.

I hope this explanation clears things up! This method is **extremely efficient** for cases where you need dynamic method invocation, but you also want to avoid the overhead of reflection on every call.