Polymorphism

Employee Example

```
public class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }
    public decimal Salary { get; set; }

    // Main constructor
    public Employee(string name, int age, decimal salary)
    {
        Name = name;
        Age = age;
        Salary = salary;
        Console.WriteLine("Full constructor called");
    }

    // Calls main constructor with default salary
    public Employee(string name, int age) : this(name, age, 0)
    {
        Console.WriteLine("Two-parameter constructor called");
    }

    // Calls two-parameter constructor with default age
    public Employee(string name) : this(name, 0)
    {
        Console.WriteLine("One-parameter constructor called");
    }

    // Calls one-parameter constructor with default name
    public Employee() : this("Unknown")
    {
        Console.WriteLine("Default constructor called");
    }
}

// Usage:
var emp = new Employee("Ahmed", 25, 5000);
// Output:
// Full constructor called

var emp2 = new Employee("Sara", 30);
// Output:
// Full constructor called
// Two-parameter constructor called

var emp3 = new Employee("Omar");
// Output:
// Full constructor called
// Two-parameter constructor called
// One-parameter constructor called
```

Let's Trace

```
public class Animal
{
    public string Name { get; set; }

    public Animal(string Name){
        this.Name = Name;
    }

    public void MakeSound1()
    {
        System.Console.WriteLine("Generic Sound");
    }

    public virtual void MakeSound2()
    {
        System.Console.WriteLine("Generic Sound");
    }

    public void Sleep()
    {
        System.Console.WriteLine($"{Name} is Sleeping");
    }

    public void Eat()
    {
        System.Console.WriteLine($"{Name} is Eating");
    }
}

public class Dog : Animal
{
    public Dog(string name) : base(name) { }

    public void MakeSound1()
    {
        System.Console.WriteLine("Fuck u im dog");
    }

    public override void MakeSound2()
    {
        System.Console.WriteLine("Fuck u im overrided dog");
    }
}

public class Program
{
    static void Main()
    {
        // First Way
        Animal animal = new Animal("B7R");
        animal.MakeSound1();
        animal.MakeSound2();
        animal.Sleep();
        animal.Eat();

        // Second Way
        Animal dogAnimal = new Dog("Mekky");
        dogAnimal.MakeSound1();
        dogAnimal.MakeSound2();
        dogAnimal.Sleep();
        dogAnimal.Eat();
```

```
        // Third Way
        Dog dog = new Dog("Riks");
        dog.MakeSound1();
        dog.MakeSound2();
        dog.Sleep();
        dog.Eat();
    }
}
```

🖊 Reference Type vs Object Type

```
Animal dogAnimal = new Dog("Mekky");
//  ↑                  ↑
// Reference Type   Object Type
```

✧ **Reference Type** ⟶ نوع الـ Variable

✧ **Object Type** ⟶ نوع الـ Object الفعلي في الـ Memory

⚠ The main difference between `MakeSound1()` and `MakeSound2()`

```
// Animal Class:
public void MakeSound1()          // Normal - Not virtual
public virtual void MakeSound2()  // virtual

// Dog Class:
public void MakeSound1()          // Method Hiding
public override void MakeSound2() // Method Overriding
```

✧ `MakeSound1()` ⟶ Compile-time Binding (Early Binding)

✧ الـ Compiler بيختار الـ Method على أساس **Reference Type**

✧ `MakeSound2()` ⟶ Runtime Binding (Late Binding)

✧ الـ CLR بيختار الـ Method على أساس **Object Type**

Analysis of the 3 methods

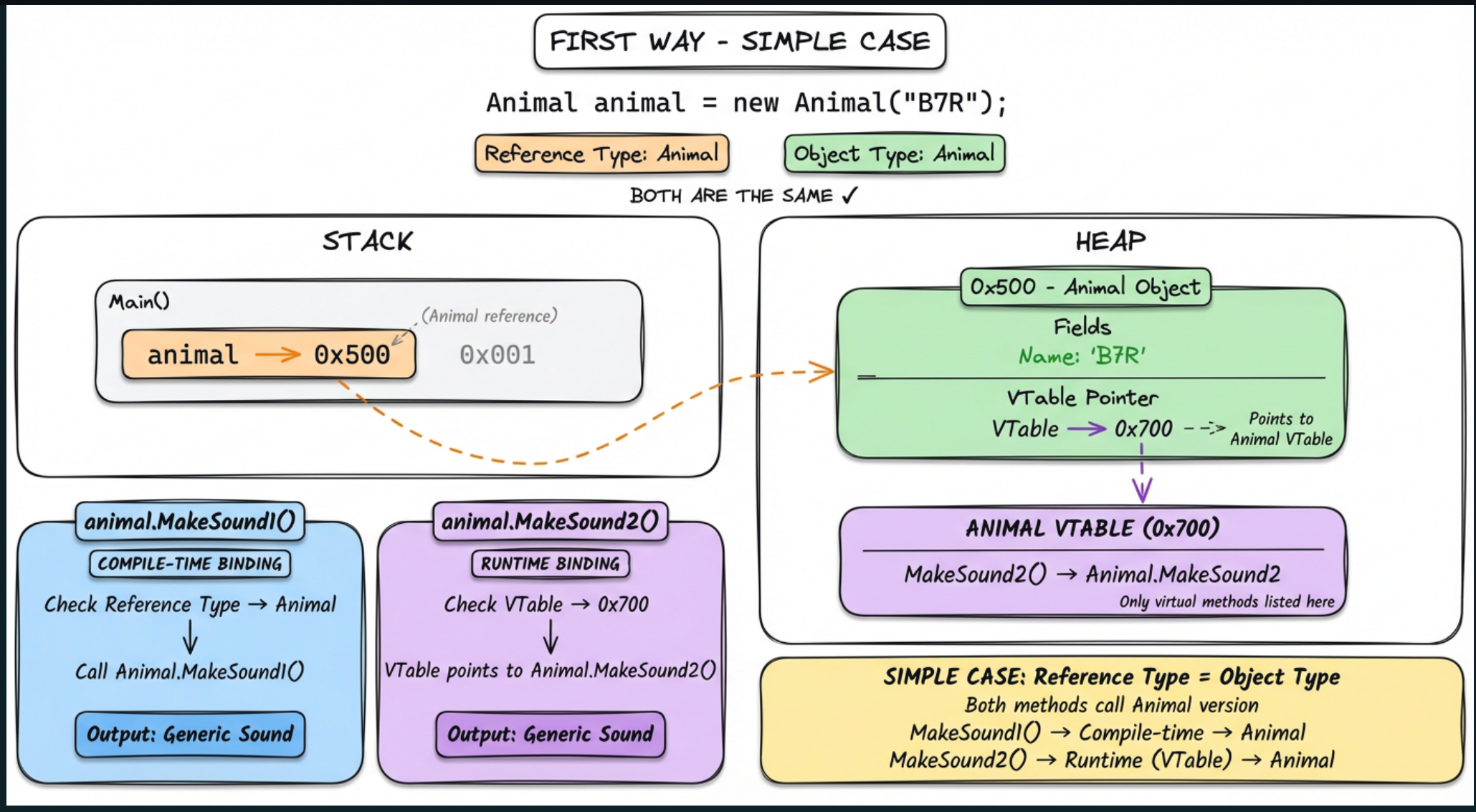
📋 First Way

```
  Animal animal = new Animal("B7R");
//  ↑      ↑           ↑
//  Type   Var      Object Type
// Animal animal      Animal
```

✧ Reference Type = **Animal**

✧ Object Type = **Animal**

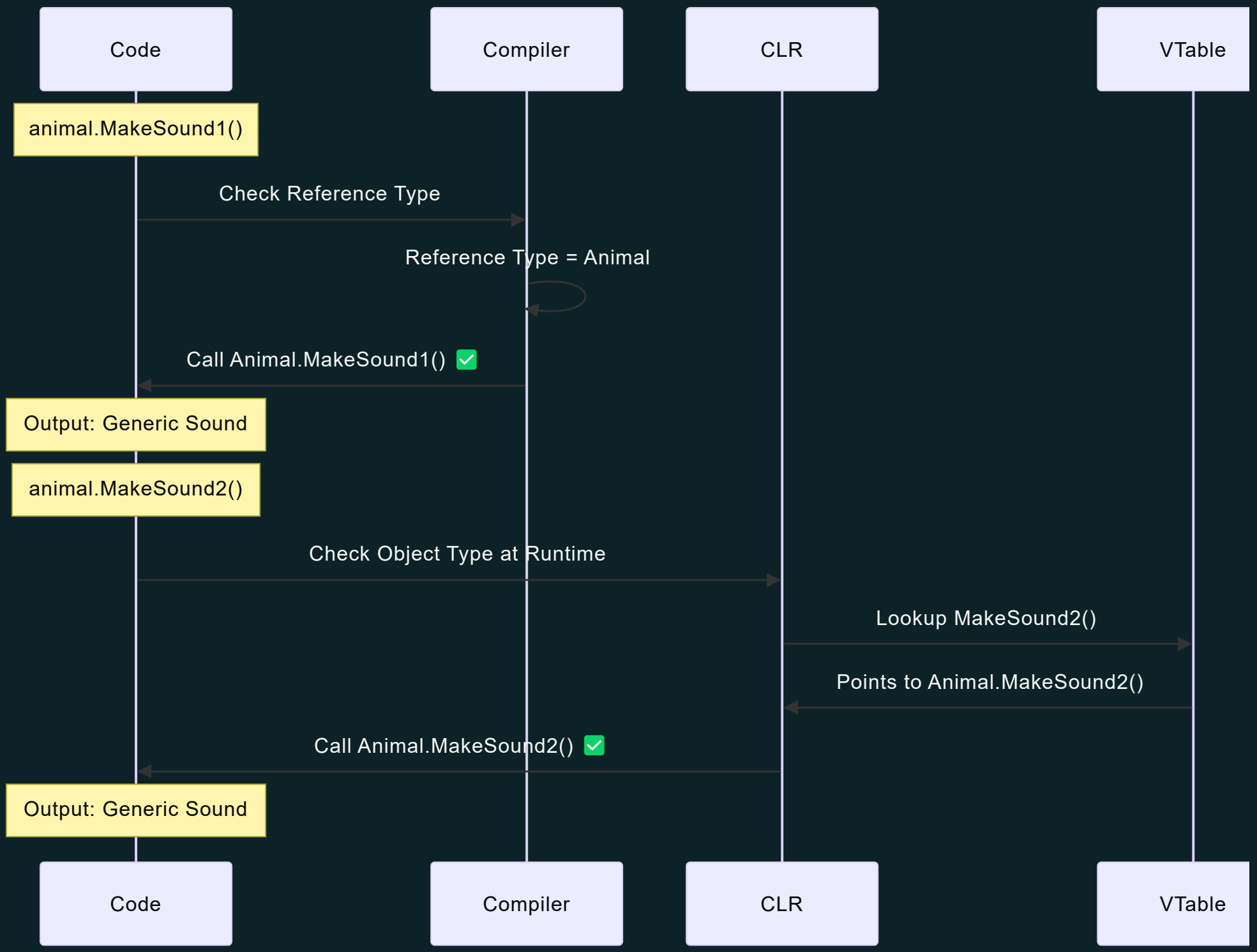
Memory Layout:



Method Calls:

```
animal.MakeSound1();  // Output: Generic Sound
animal.MakeSound2();  // Output: Generic Sound
```

Why?



in-short

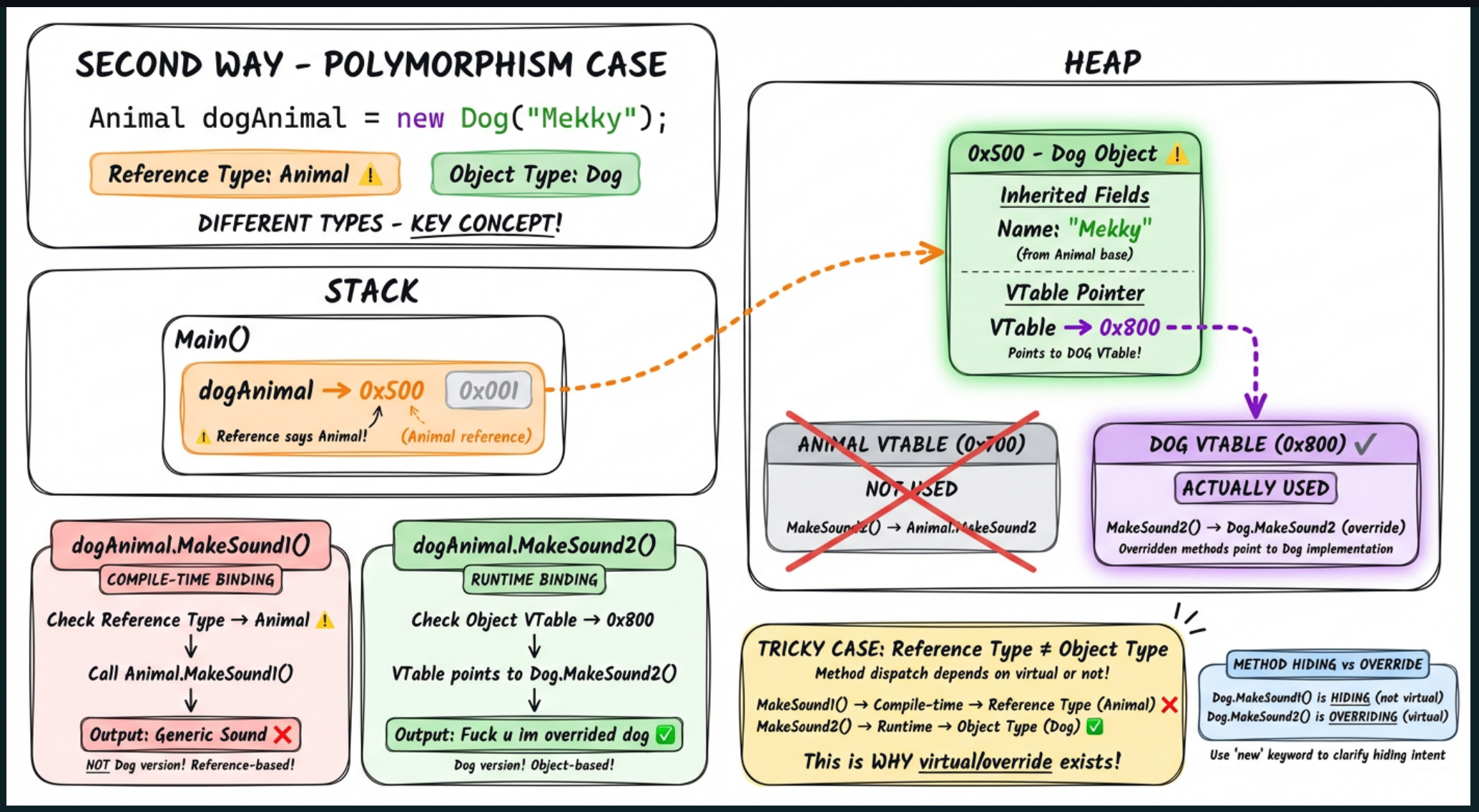
- ✧ **MakeSound1()** → Compile-time → Animal version
- ✧ **MakeSound2()** → Runtime → Animal version (من الـ VTable)

Second Way

```
Animal dogAnimal = new Dog("Mekky");
//  ↑      ↑           ↑
//  Type   Var      Object Type
// Animal dogAnimal   Dog
```

- ✧ Reference Type = **Animal**
- ✧ Object Type = **Dog**

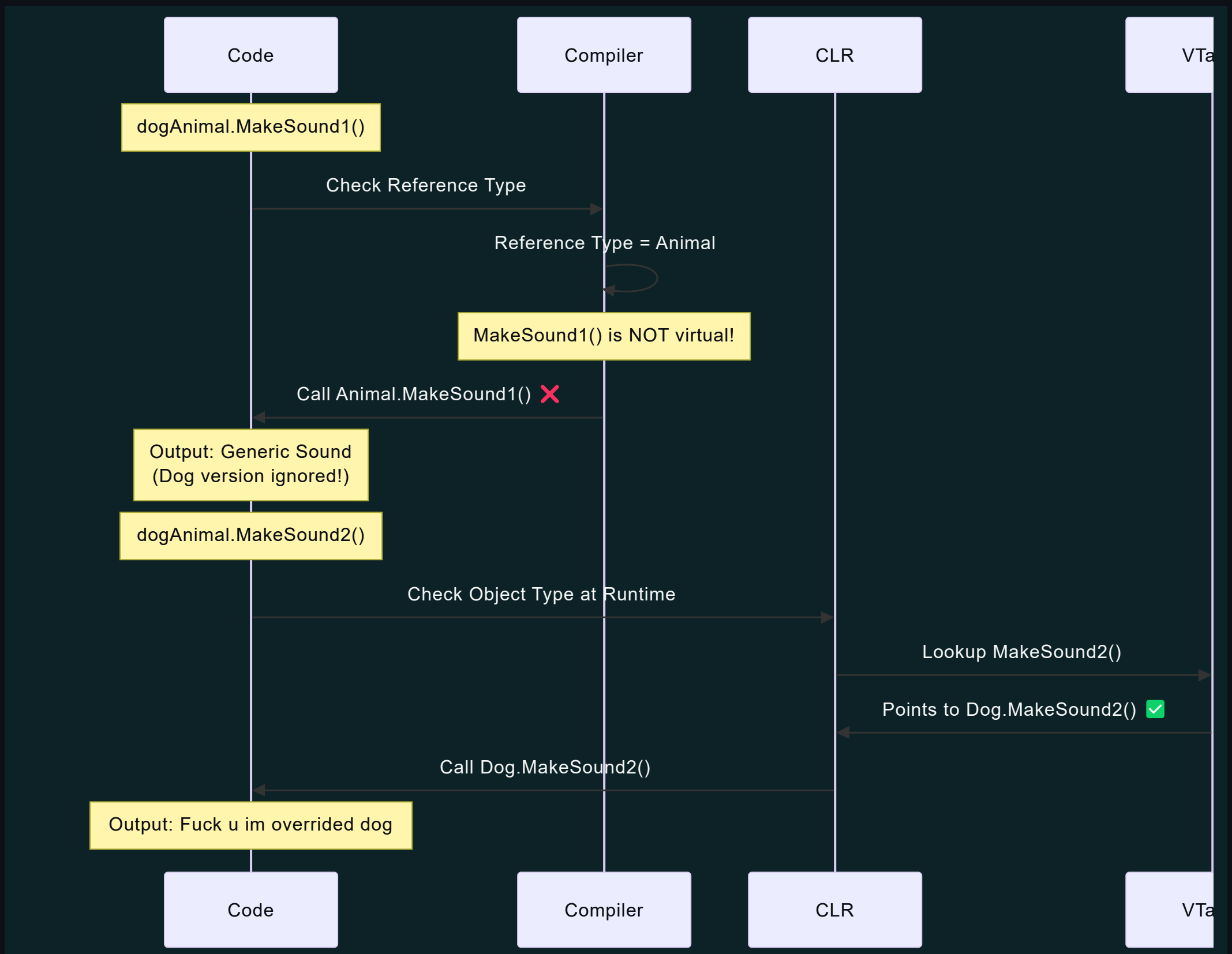
Memory Layout:



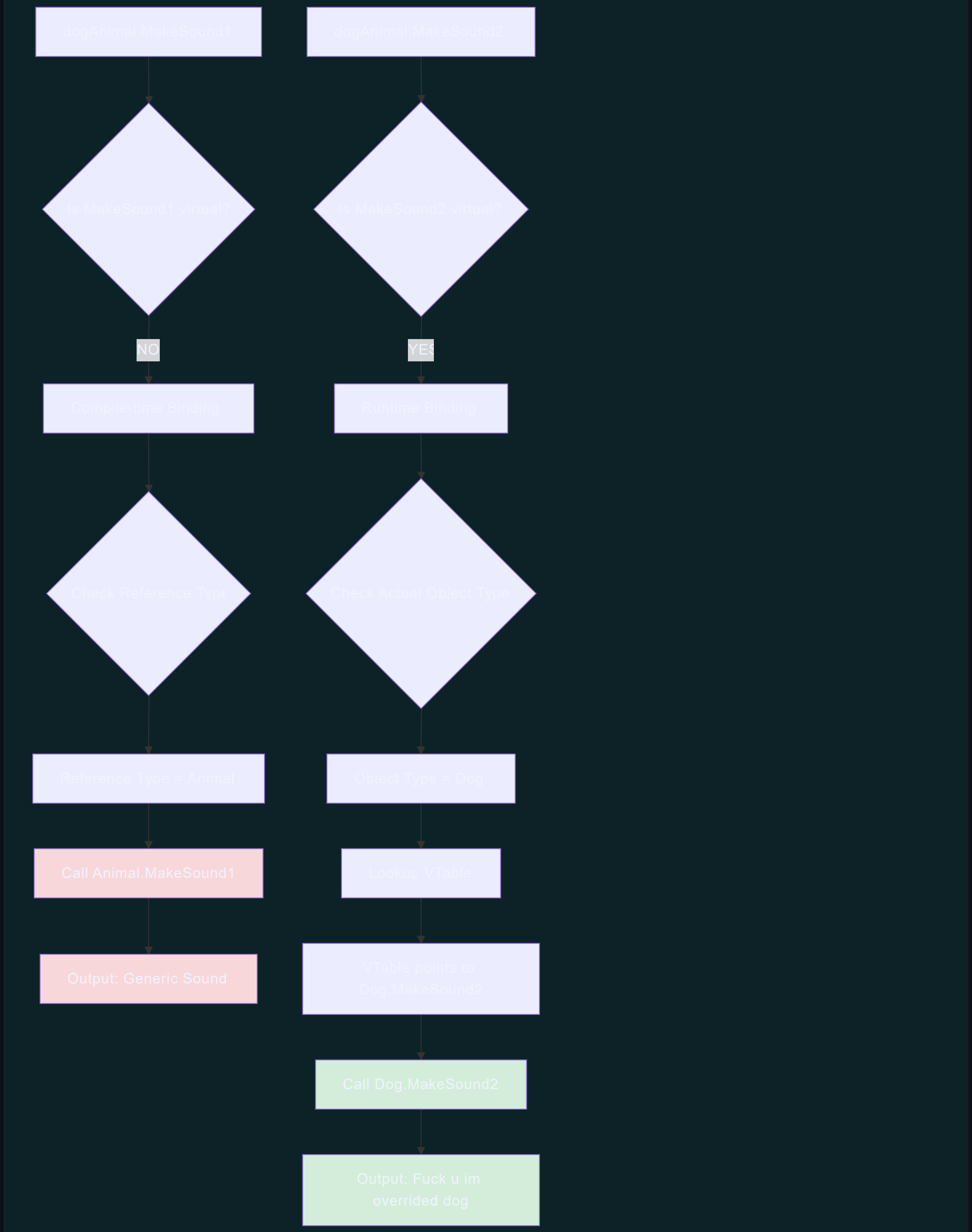
Method Calls

```
dogAnimal.MakeSound1();  // Output: Generic Sound
dogAnimal.MakeSound2();  // Output: Fuck u im overrided dog
```

Why?


Code
Compiler
CLR
VTa
dogAnimal.MakeSound1()
Check Reference Type
Reference Type = Animal
MakeSound1() is NOT virtual!
Call Animal.MakeSound1() ❌
Output: Generic Sound
(Dog version ignored!)
dogAnimal.MakeSound2()
Check Object Type at Runtime
Lookup MakeSound2()
Points to Dog.MakeSound2() ✅
Call Dog.MakeSound2()
Output: Fuck u im overrided dog
Code
Compiler
CLR
VTa

The Flow



⚠ Tricky Part!

```
// Dog class:
public void MakeSound1()
```

انت لما بتعملها كد من غير `virtual` , `override` بنسبة كبيره بيبقي الهدف انك تـ Hide ال Method بتاعت الـ Parent Class ولاكن الـ Compiler مبيبقاش فاهم ده لوحده، وعشان توضحله نيتك اكتر بنحط ال Keyword `new`

```
public new void MakeSound1()  // ✅ Clearly, you want to hide the parent method
```

in-short

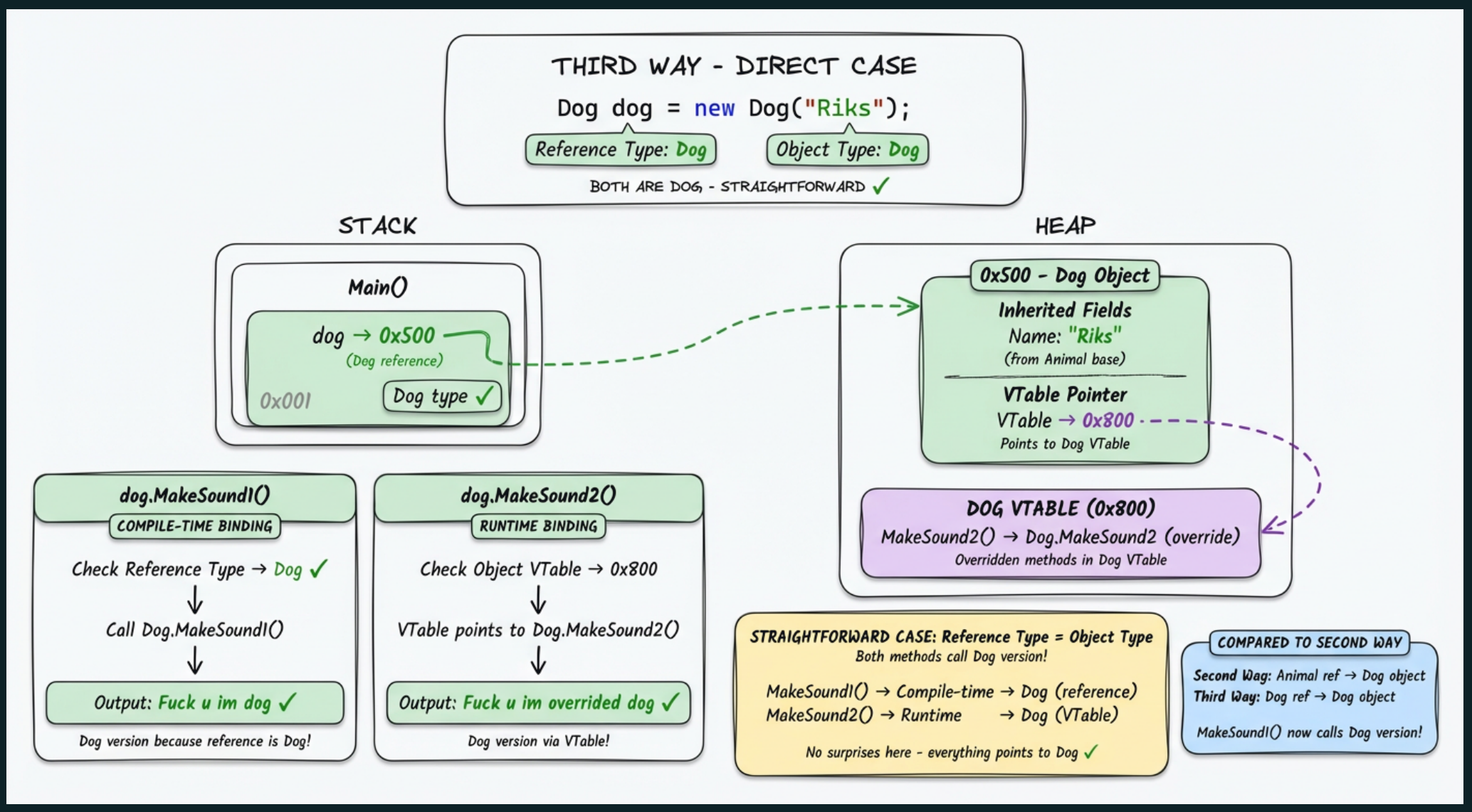
✧ **MakeSound1()** → Compile-time → Reference Type → Animal version
✧ **MakeSound2()** → Runtime → Object Type → Dog version

Third Way

```
Dog dog = new Dog("Riks");
//  ↑   ↑       ↑
// Type Var  Object Type
// Dog  dog    Dog
```

✧ Reference Type = **Dog**
✧ Object Type = **Dog**

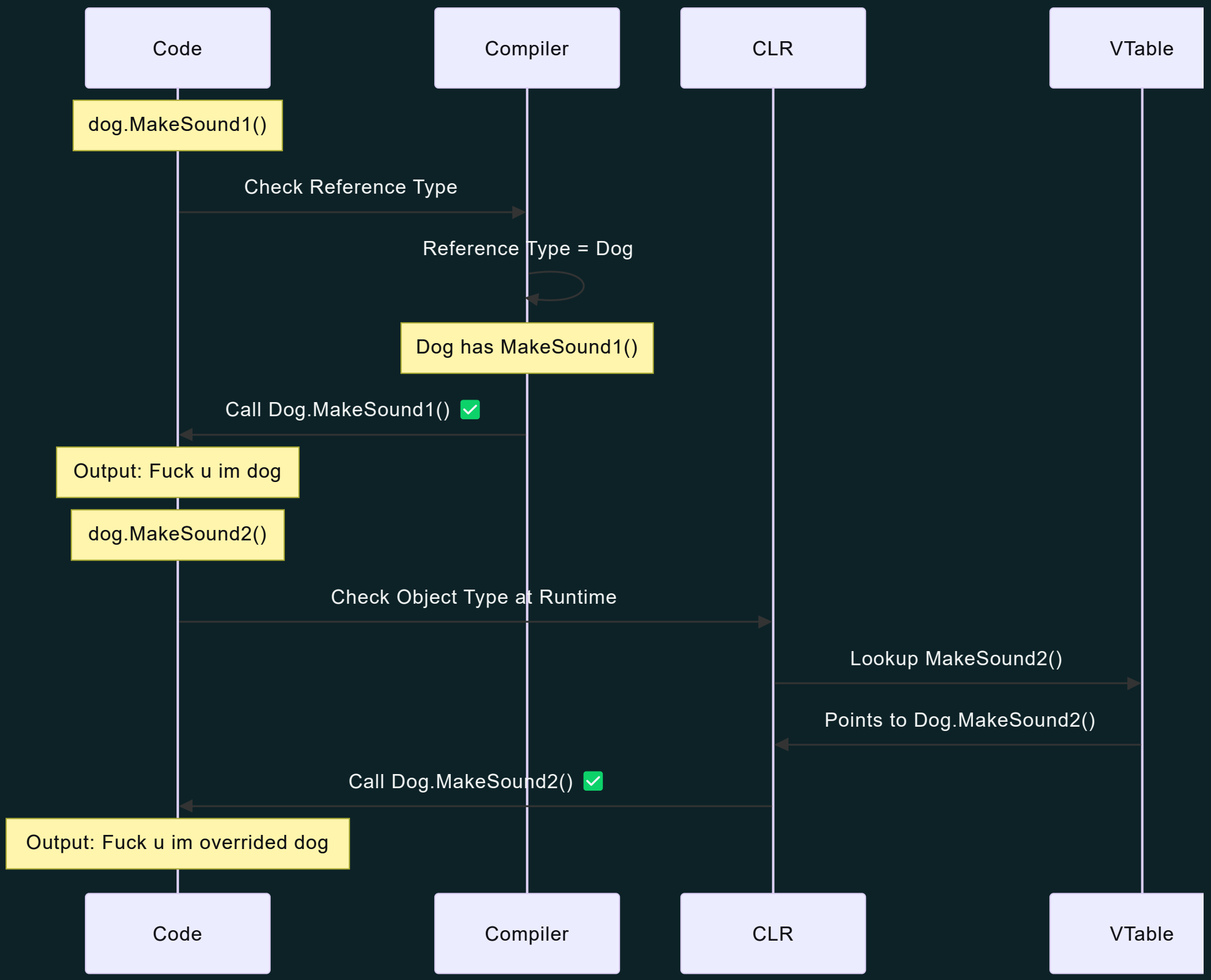
Memory Layout:



Method Calls:

```
dog.MakeSound1();  // Output: Fuck u im dog
dog.MakeSound2();  // Output: Fuck u im overrided dog
```

Why?



in-short

- ✧ **MakeSound1()** → Compile-time → Reference Type → Dog version
- ✧ **MakeSound2()** → Runtime → Object Type → Dog version

Summary

WAY	REFERENCE TYPE	OBJECT TYPE	MAKESOUND1()	MAKESOUND2()
#1 `Animal animal = new Animal()`	Animal	Animal	Animal.MakeSound1() ***(Compile-time)***	Animal.MakeSound2() ***(Runtime)***
#2 `Animal dogAnimal = new Dog()`	Animal ⚠️	Dog	Animal.MakeSound1() ❌ ***(Compile-time - Reference Based!)***	Dog.MakeSound2() ✅ ***(Runtime - Object Based!)***
#3 `Dog dog = new Dog()`	Dog	Dog	Dog.MakeSound1() ***(Compile-time)***	Dog.MakeSound2() ***(Runtime)***

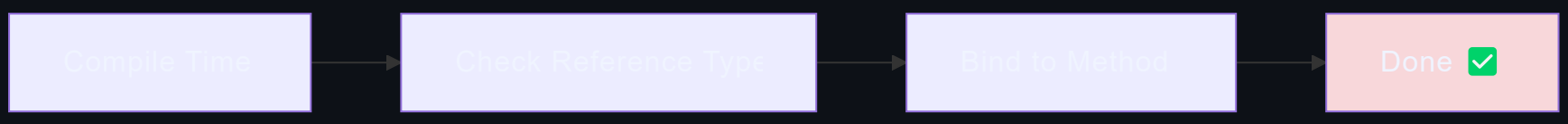
الخلاصة

✧ هشوف الـ Method فالـ Classes:

✧ هل هيـ **General** Methods: ف دلوقتي ال **Compiler** قدر يحدد فالـ **Compile-time** ان الـ Method اللي هتتنفذ هي **RefType.Method**

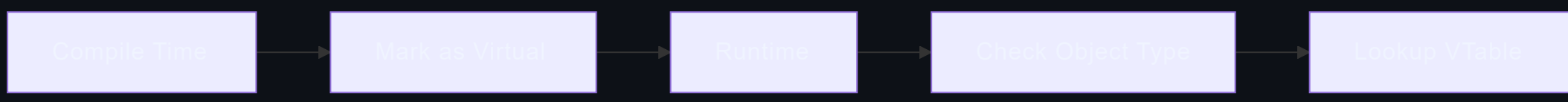
✧ هل هيـ **Virtual/Override** Methods: ف دلوقتي ال **CLR** هيحدد فالـ Runtime ان الـ Method اللي هتتنفذ هي **ObjType.Method**

Non-Virtual (No VTable):



Fast, but **not flexible**

Virtual (With VTable):



Slightly slower, but **polymorphic**

VTable (Virtual Method Table)

What is a VTable?

✧ هو **جدول في الـ Memory** بيتعمل لكل Class فيه Methods Virtual

✧ بيكون فيه **Pointers** للـ Virtual Methods

✧ بيستخدمه الـ **CLR** عشان يعرف يوصل للـ Method الصحيحة في الـ **Runtime**

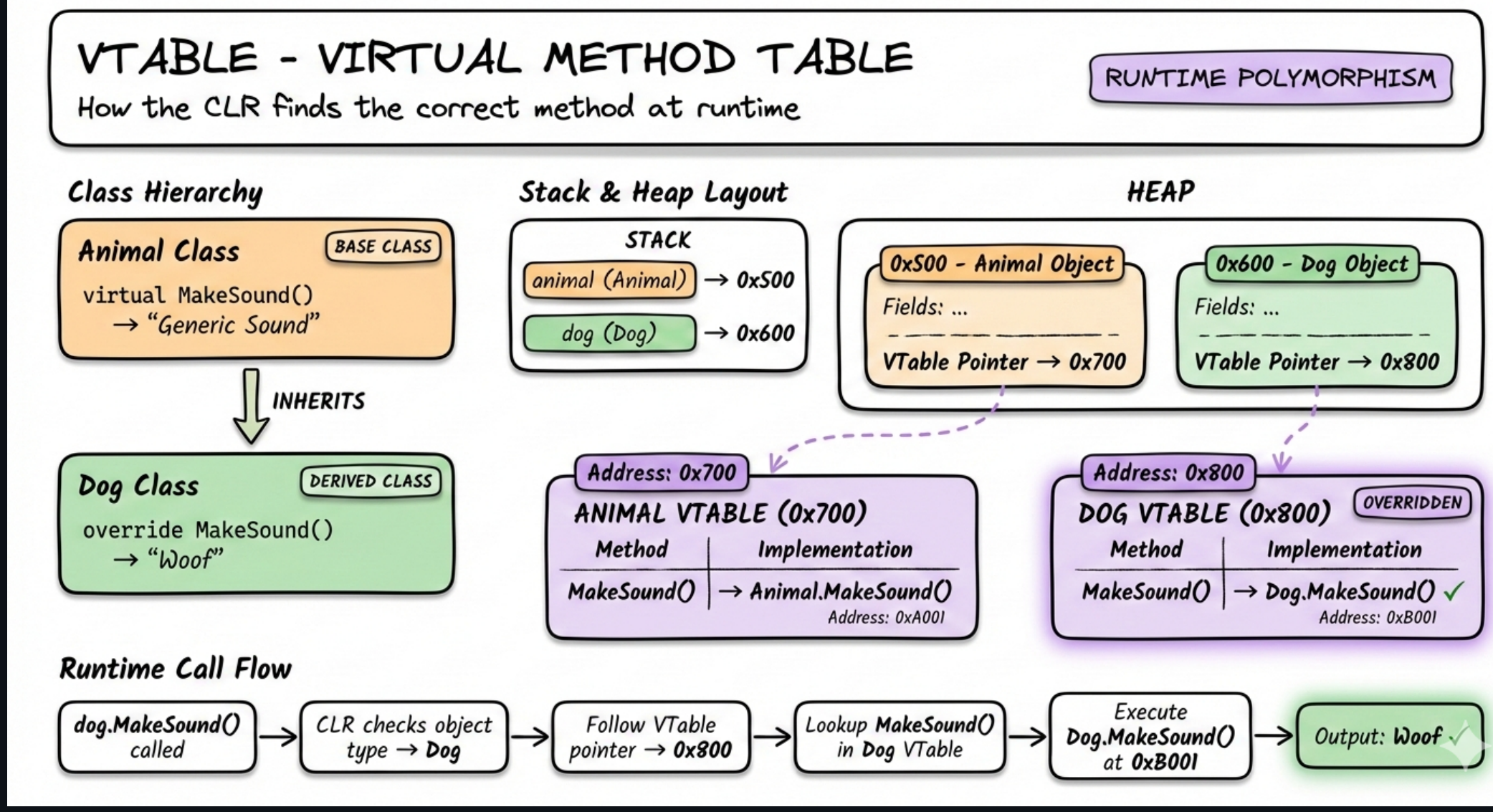
✧ بمعني اخر: بيقول الجدول ده بيقول للـ CLR لو حد عمل Call لـ Virtual Method، اروح انفذ انهي Implementation بالظبط ؟

Example VTable

```
public class Animal
{
    public virtual void MakeSound()  // virtual
    {
        Console.WriteLine("Generic Sound");
    }
}

public class Dog : Animal
{
    public override void MakeSound()  // override
    {
        Console.WriteLine("Woof");
    }
}
```

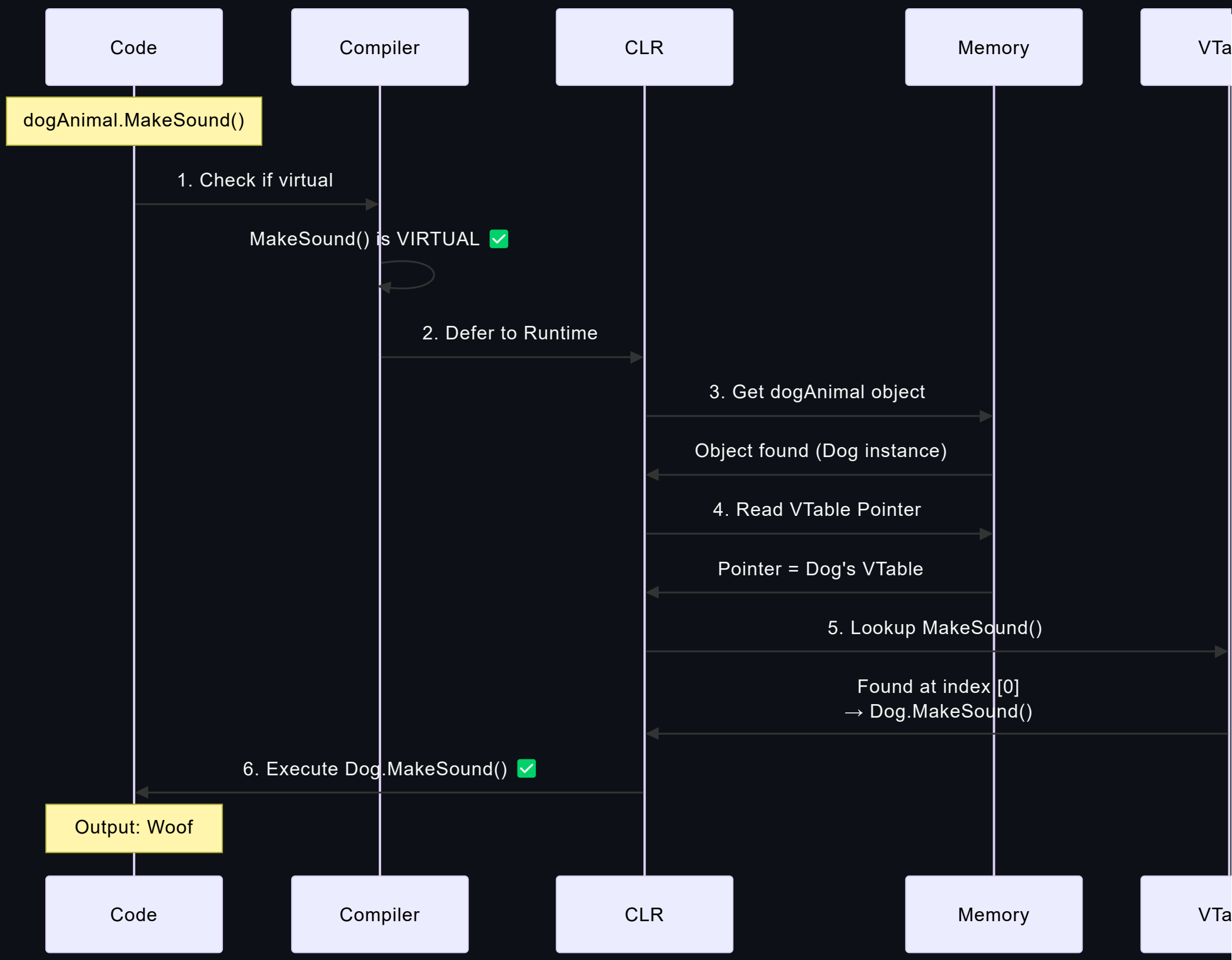
Memory Layout with VTable:



كل **Object** فيه **Pointer** بيشاور على الـ **VTable** بتاع الـ **Class** بتاعه

✧ لما بتنادي على Virtual Method، الـ CLR بيروح للـ VTable ويلاقي العنوان الصحيح

The Flow:

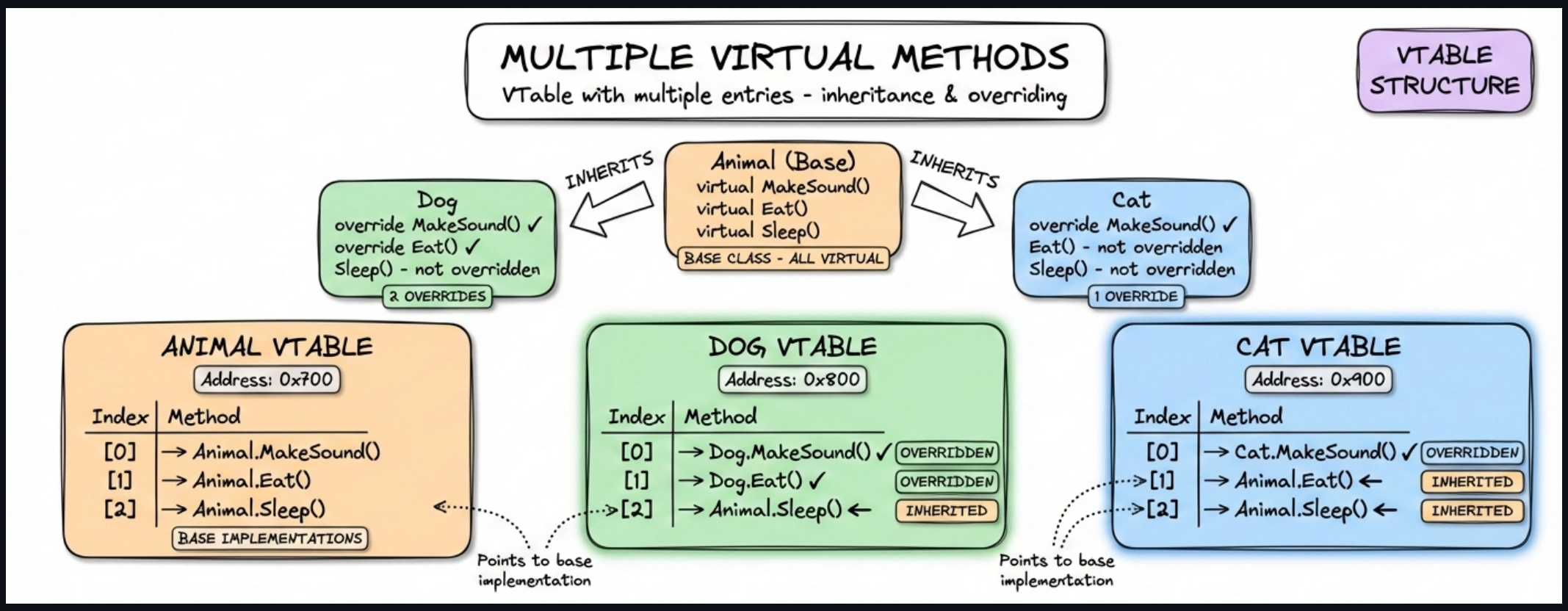


Multiple Virtual Methods

```
public class Animal
{
    public virtual void MakeSound() { }
    public virtual void Eat() { }
    public virtual void Sleep() { }
}

public class Dog : Animal
{
    public override void MakeSound() { }
    public override void Eat() { }
    // Sleep() مش overridden
}

public class Cat : Animal
{
    public override void MakeSound() { }
    // Eat() و Sleep() مش overridden
}
```



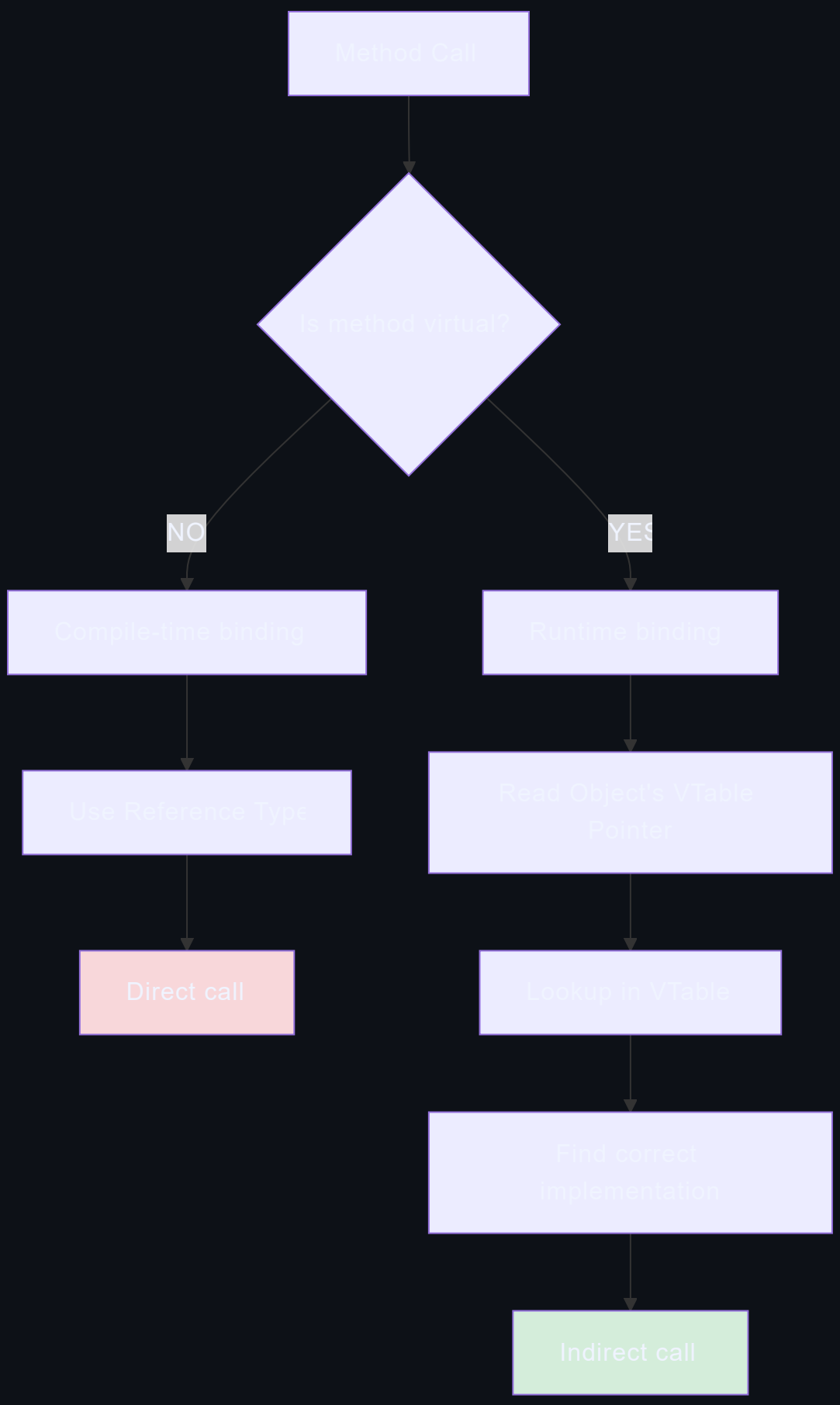
Notice That

- ✧ لو Method مش Overridden في الـ Child → الـ VTable بيشاور على **Parent's implementation**
- ✧ الـ VTable Indices بتكون **متطابقة** في كل الـ Classes (نفس الترتيب)

VTable Creation Timeline

- ✧ الـ **VTable** بيتعمل **مرة واحدة** لكل Class فيه Virtual / Override Methods
- ✧ يعني هيعملك VTable للـ Class اللي فيه Virtual Method و VTable مختلف تماما للـ Class اللي فيه Override Method
- ✧ بعد كدا اي Object هتعمله من ال 2 Classes دول هيبقى شايل Pointer بيشاور علي VTable بتاع ال Class اللي هوا منه
- ✧ كل الـ **Objects** من نفس الـ Classes بيشاوروا على **نفس الـ VTable**

Decision Tree:



When to Use Abstract Class?

🔥 Use Abstract Class When:

✧ Define an IS-A relationship

```
// Dog IS-A Animal
public abstract class Animal { }
public class Dog : Animal { }
```

✧ Want to share code between classes

```
public abstract class Repository
{
    // Shared code
    protected void LogQuery(string query)
    {
        Console.WriteLine($"Query: {query}");
    }
}
```

✧ Need fields or non-public members

```
public abstract class BaseService
{
    protected string _connectionString; // Field
    private int _retryCount = 3;        // Private field
}
```

✧ Need constructor logic

```
public abstract class Entity
{
    public Entity()
    {
        CreatedAt = DateTime.Now;
    }
}
```

When to Use Interface?

Use Interface When:

✧ Define a CAN-DO capability

```
// A car CAN-BE driven
public interface IDrivable { }
public class Car : IDrivable { }
```

✧ Need multiple inheritance

```
public class SmartPhone : ICallable, IMessagable, IConnectable
{
    // Can implement multiple interfaces
}
```

✧ Only want to define a contract

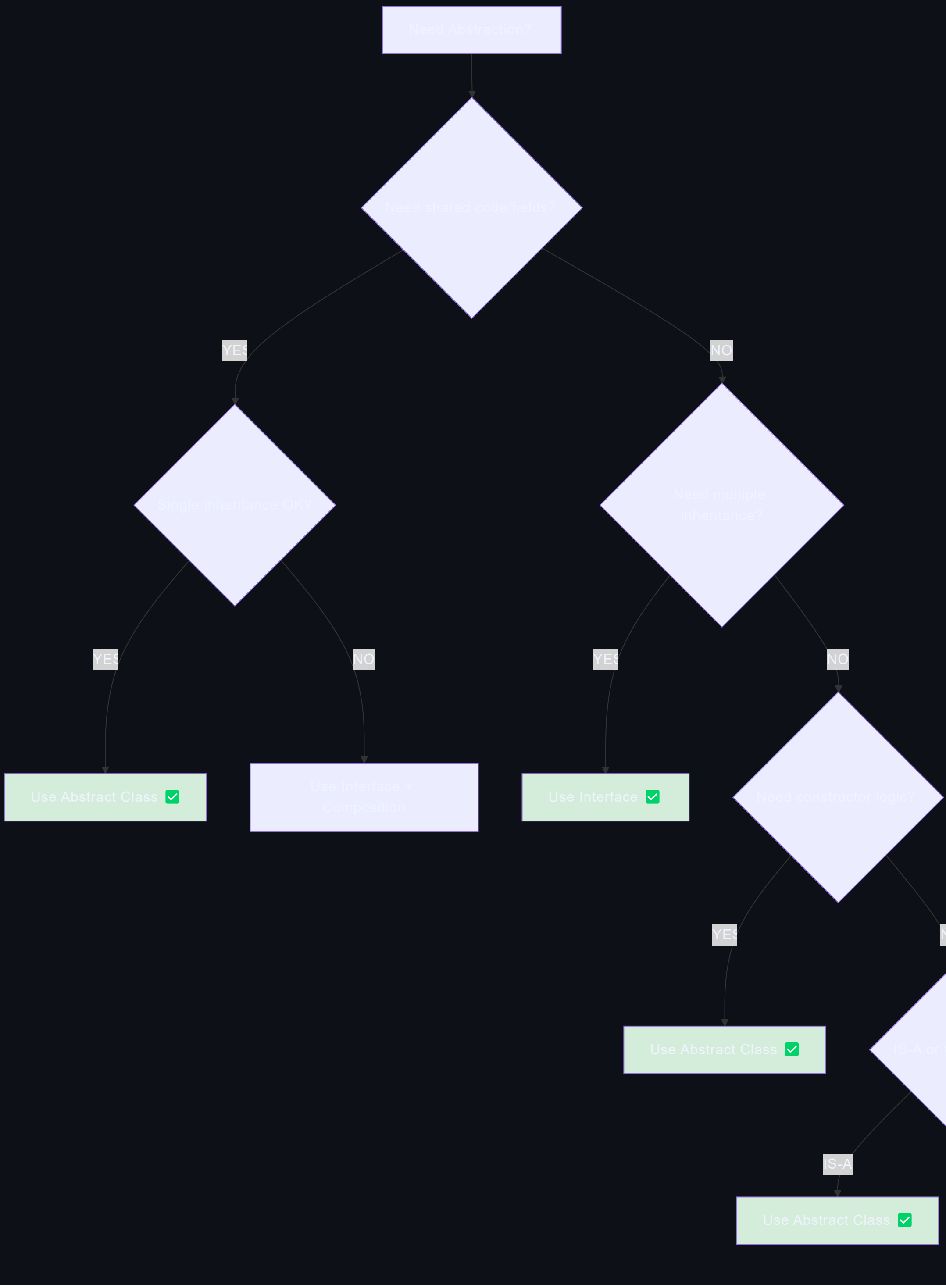
```
public interface IRepository<T>
{
    void Add(T item);
    T Get(int id);
    void Delete(int id);
}
```

✧ Want to use Dependency Injection

```
public interface IEmailService { }
public class SmtpEmailService : IEmailService { }
public class SendGridEmailService : IEmailService { }

// Easy to swap implementations
```

Abstract vs Interface Decision Tree


Need Abstraction?
Need shared code/fields?
YES
NO
Single inheritance OK?
YES
NO
Use Abstract Class
Use Interface + Composition
Need multiple inheritance?
YES
NO
Use Interface
Need constructor logic?
YES
NO
Use Abstract Class
IS-A
Use Abstract Class