Dynamic Method Dispatch

Dynamic method dispatch in Kotlin refers to the mechanism by which a call to an overridden method is resolved at runtime, rather than at compile time. This is a core concept in object-oriented programming that allows for achieving **runtime polymorphism**.

It is the foundation of **runtime polymorphism**, which allows methods to behave differently based on the actual object that is instantiated, even if the reference variable is of a superclass type. This behavior is also known as **late binding** or **dynamic binding**, as the method to be executed is determined during runtime.

Key Concept:

- **Polymorphism** allows a class to perform a single action in different ways.

- **Inheritance:**
  - In Kotlin, inheritance is a mechanism where one class (child class or subclass) inherits properties and behaviors (methods) from another class (parent class or superclass).
  - A subclass can **override** methods of the parent class to provide its own implementation.

- **Method Overriding:**
  - Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.
  - The overridden method in the subclass must have the same method signature (name, return type, parameters) as in the superclass.
  - In Kotlin, you must use the `open` keyword to make a method in the superclass overridable, and the `override` keyword in the subclass to override it.

- **Reference Type vs. Object Type:**
  - **Reference type:** This is the type of the variable that holds the reference to the object (e.g., `Animal` in the earlier example).
  - **Object type:** This is the actual type of the object that the reference variable refers to (e.g., `Dog` or `Cat`).
  - Dynamic method dispatch works by deciding which method to call based on the object type, not the reference type, during runtime.

- **Superclass Reference can't call any specialized Method of its child class :**
  - No, you **cannot directly call specialized methods or members** that are only present in a subclass (or child class) using a reference of the superclass, even if the object being referred to is of the subclass.
  - This is because a superclass reference only has knowledge of the members (methods and properties) that are defined in the superclass. It does not have access to any specialized methods or properties that are unique to the subclass.
  - In order to access those specialized methods or properties, you need to **explicitly cast** the superclass reference to the subclass type.

```
fun main() { val animal: Animal = Dog()

(animal as Dog).wagTail()// Explicit cast to Dog, now you can call wagTail() }
```

How Dynamic Method Dispatch Works:

Superclass Method Declaration:

- The superclass declares a method using the `open` keyword. This means the method can be overridden in subclasses.

Subclass Method Overriding:

- The subclasses override the `sound()` method using the `override` keyword to provide their specific implementations.

Reference and Object Assignment:

- A reference variable of type `Animal` can refer to an object of type `Dog`, `Cat`, or any other subclass of `Animal`. This is possible because of inheritance.

```
val animal: Animal = Dog()  // Reference type is Animal, but the
                        // object type is Dog
```

Method Call During Runtime:

- When the method `sound()` is called on the reference `animal`, Kotlin checks the actual type of the object during runtime (in this case, `Dog`), and invokes the method implementation of the object type (`Dog`'s `sound()` method).
- This decision to call the overridden method happens dynamically at runtime. And so called Dynamic Method dispatch.

Example with More Context

Let's assume you are designing a system that simulates a zoo. You have different types of animals, and you want to simulate their sounds. You can use dynamic method dispatch to make your system flexible and scalable.

```
kotlin
Copy code
open class Animal {
    open fun sound() {
        println("The animal makes a sound")
    }
}

class Lion : Animal() {
    override fun sound() {
        println("The lion roars")
    }
}


class Elephant : Animal() {
    override fun sound() {
        println("The elephant trumpets")
    }
}
```

```
fun makeAnimalSound(animal: Animal) {
    animal.sound()  // This line will dynamically call the correct method based on the
object type
}

fun main() {
    val lion: Animal = Lion()
    val elephant: Animal = Elephant()

    makeAnimalSound(lion)       // Outputs: "The lion roars"
    makeAnimalSound(elephant)   // Outputs: "The elephant trumpets"
}
```

Why This is Useful (Benefits of Dynamic Method Dispatch)

1. **Polymorphism:**
    - Dynamic method dispatch enables **polymorphism**, where a single reference type (like `Animal`) can refer to objects of different types (like `Dog` and `Cat`), and the method behavior varies accordingly.
    - You can write more flexible and reusable code using polymorphism, as the same code can work with different types of objects.
2. **Loose Coupling:**
    - The code that calls the method (e.g., `animal.sound()`) is loosely coupled to the actual class of the object (`Dog`, `Cat`, etc.). This allows you to change the underlying object type without changing the code that uses it.
3. **Maintainability:**
    - If you add more subclasses in the future (e.g., `Lion`, `Elephant`), the same polymorphic code can handle these new types without needing modification. This reduces code duplication and increases maintainability.

Solution: Calling Specialized Methods < Type Casting >

To access the `wagTail()` method, you need to cast the `Animal` reference back to `Dog`:

Using Smart Casts in Kotlin:

Kotlin supports **smart casting**, so if you check the type at runtime, you can safely cast it to the subclass:

```
kotlin
Copy code
fun main() {
    val animal: Animal = Dog()  // Reference type is Animal, object type is Dog

    // Check if the reference is actually a Dog
    if (animal is Dog) {
        animal.wagTail()  // Smart cast: Compiler knows it's a Dog, so now you can call
wagTail()
    }
}
```

In the `if (animal is Dog)` check, Kotlin automatically casts the reference `animal` to type `Dog`, allowing you to call the `wagTail()` method.

Explicit Type Casting:

If you know that the object referred to by the superclass reference is actually an instance of the subclass, you can explicitly cast it using the `as` keyword:

```
kotlin
Copy code
fun main() {
    val animal: Animal = Dog()

    (animal as Dog).wagTail()  // Explicit cast to Dog, now you can call wagTail()
}
```

If `animal` is not actually an instance of `Dog` and you try to cast it, Kotlin will throw a `ClassCastException` at runtime. To avoid this, you can use the safe cast operator `as?`:

```
kotlin
Copy code
fun main() {
    val animal: Animal = Dog()

    val dog = animal as? Dog  // Safe cast, returns null if casting fails
    dog?.wagTail()  // Only calls wagTail() if the cast was successful.
```