Interface

In Kotlin, an **interface** is a blueprint of a class that can contain **abstract methods** (methods without implementation) and **properties**. Unlike abstract classes, interfaces cannot store state (i.e., they don't have instance fields). Interfaces are often used to define common behavior across different classes that may not share a common parent class.

1. Defining an Interface

You define an interface using the `interface` keyword. For example:

```
kotlin
Copy code
interface Drivable {
    fun drive()  // Abstract method
}
```

This `Drivable` interface has a single abstract method `drive()` that any class implementing the interface must provide.

2. Implementing an Interface

A class implements an interface using the `:` `(colon)` syntax. The class must override all abstract methods in the interface.

```
kotlin
Copy code
class Car : Drivable {
    override fun drive() {
        println("Car is driving")
    }
}
```

The `Car` class implements the `Drivable` interface and provides the implementation for the `drive()` method.

3. Multiple Interfaces

A class in Kotlin can implement multiple interfaces. If the same method is defined in multiple interfaces, the implementing class must resolve the conflict by overriding the method.

```
kotlin
Copy code
interface Flyable {
    fun fly()
}

class FlyingCar : Drivable, Flyable {
    override fun drive() {
        println("Flying Car is driving")
    }

    override fun fly() {
        println("Flying Car is flying")
    }
}
```

Here, `FlyingCar` implements both `Drivable` and `Flyable`, and provides implementations for both `drive()` and `fly()` methods.

4. Default Implementations in Interfaces

Kotlin interfaces can provide **default method implementations**, unlike in some other programming languages. This means that classes implementing the interface don't have to override the methods that have a default implementation unless they want to change the behavior.

```
kotlin
Copy code
interface Steerable {
    fun steer() { // concrete method "default method"
        println("Default steering")
    }
}

class Bike : Steerable {
    // Uses the default implementation of steer()
}
```

In the example above, the `Bike` class inherits the default implementation of the `steer()` method from the `Steerable` interface without having to override it.

5. Properties in Interfaces

Interfaces can also define properties, but they cannot store values. The implementing class must provide a value or implementation for the property. Properties in an interface can't have state, they must be abstract.

```
kotlin
Copy code
interface Identifiable {
    val id: String        // Abstract Property declaration
    val password: String // Abstract Property declaration
}

class User : Identifiable {

    override val id: String = "tcs_5683" // overriding property by direct initialization

    // we can also override by setting what get method will returns
    override val password: String
        get() = "x453p89023"

}
```

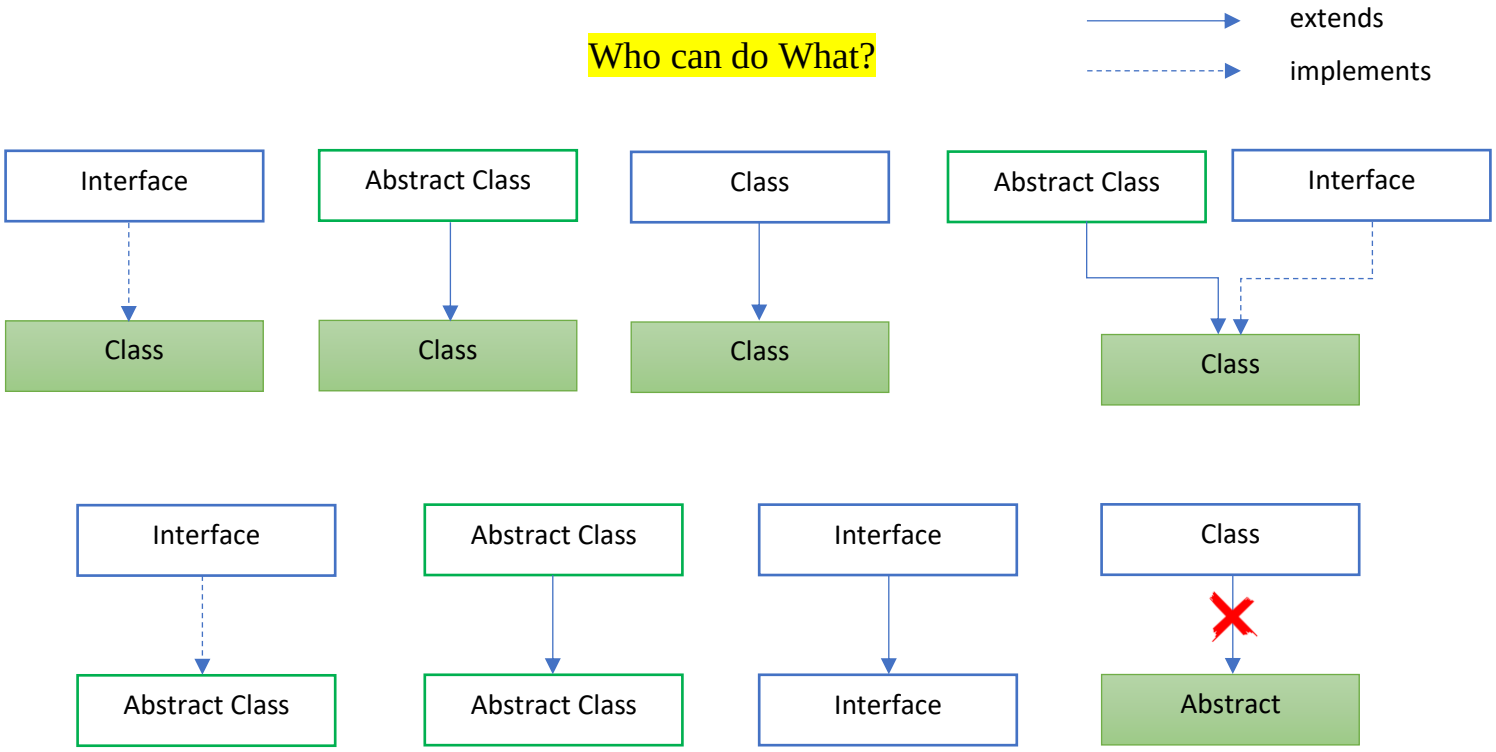
Here, the `User` class implements the `Identifiable` interface and provides a value for the `id` property.

Example with Multiple Interfaces and Default Methods

```
kotlin
Copy code
interface Walkable {
//  val speed: Int = 2 Not allowed
    fun walk() {
        println("Walking by default")
    }
}

interface Runnable {
    fun run()
}

class Robot : Walkable, Runnable {
    override fun run() {
        println("Robot is running")
    }
}

fun main() {
    val robot = Robot()
    robot.walk()  // Uses default method from Walkable
    robot.run()   // Uses overridden method in Robot
}
```

Note – If the same interface Runnable also contains walk method with body, then it will create an ambiguity in the child class implementing both interfaces, and we have to manually resolve this conflict by overriding the walk() method in child class so only one body exists.

However, if abstract run() method also present in Walkable interface then it will not create problem. Since these each of walk() method present in own interface, without any body, and their actual implementation is provided by child class overriding walk() method.



Feature	Abstract Class	Interface
Keyword	`abstract` class	`interface`
Purpose	Represents a base class for related objects, sharing state and behavior	Represents a contract of behavior that classes must implement
Methods	Can have both abstract (no implementation) and concrete (implemented) methods	Can have both abstract methods and default methods (with implementation)
State (Fields)	Can have instance variables with values.	Cannot have instance variables with values, but can have abstract properties
Multiple Inheritance	Cannot be inherited by multiple classes (single inheritance only)	Supports multiple inheritance (a class can implement multiple interfaces)
Constructors	Can have constructors and constructor parameters	Cannot have constructors
Visibility Modifiers	Methods and properties can have visibility modifiers (`private`, `protected`, `public`)	All methods and properties are `public` by default, no `protected` or `private` allowed
When to Use	When you need to share common state or partial implementation across a set of subclasses	When you need to define a contract or common behavior without sharing state
Default Implementation	Can provide concrete methods	Can provide default methods since Kotlin 1.1
Instance Creation	Cannot be instantiated directly	Cannot be instantiated directly
Inheritance	A class can extend only one abstract class	A class can implement multiple interfaces
Use of `super` keyword	Can use `super` to access superclass methods	Can use `super` to access default method implementations, but only if explicitly defined in interfaces
Properties	Can have both concrete (with default values) and abstract properties	Can have abstract properties but not fields (no default values)

Interface is only true way to achieve abstraction

Though interface we can perform multiple inheritance.

Interface = abstract properties + abstract methods + default method.

In most of cases interfaces are used to represent Polymorphism (one thing having many different forms).