Sealed Class

sealed class in Kotlin is like a special type of class that helps you organize and control different possibilities in your program. It's used when you have a fixed number of options, and you want to make sure those options are clear and handled properly.

Think of it like this:

Imagine you have a vending machine, but it only sells three things: chips, soda, and candy. The vending machine can't sell anything else, and you know exactly what it can offer. In Kotlin, this would be a **sealed class**, where you define all the items (chips, soda, candy) it can "sell."

Key Points:

1. **Limited Options**: A sealed class lets you define a limited set of options (subclasses). Nothing else can be added unless you do it in the same file.
2. **Ensured Handling**: When you use a sealed class, Kotlin makes sure you handle all the possibilities. So if you have a function to "choose" an item from the vending machine, it will ensure you cover all the options (chips, soda, and candy). This prevents mistakes like forgetting an option.
3. **Abstract Base Class**: You can't create an object directly from the sealed class itself; you can only create it from one of its subclasses (like chips, soda, or candy).

Simple Example:

```
kotlin
Copy code
sealed class VendingItem {
    class Chips(val flavor: String) : VendingItem()
    class Soda(val brand: String) : VendingItem()
    object Candy : VendingItem() // Fixed candy type
}

fun selectItem(item: VendingItem): String {
    return when (item) {
        is VendingItem.Chips -> "You selected Chips of flavor: ${item.flavor}"
        is VendingItem.Soda -> "You selected Soda of brand: ${item.brand}"
        VendingItem.Candy -> "You selected Candy"
    }
}
```

Here:

- **VendingItem** is the sealed class.
- It has three specific options: **Chips**, **Soda**, and **Candy**.
- The `when` expression ensures you handle all the possibilities (chips, soda, candy), so you can't miss any!

Why Use It?

- It makes sure you handle every case (chips, soda, candy).
- It gives you control over what types (subclasses) can exist, so no unexpected options pop up.

In short, sealed classes help you organize code where you have a limited, fixed number of possibilities, and you want to make sure all of them are accounted for!

Q) What is this statement in sealed class - object Candy : VendingItem() ?

The statement `object Candy : VendingItem()` in a sealed class defines a **singleton object** called `Candy` that is a subclass of the sealed class `VendingItem`.

Breaking it Down:

1. **`object`**: In Kotlin, `object` is used to declare a **singleton**, meaning there will only be one instance of `Candy` in your program. You can't create multiple instances of `Candy`. There will be exactly one `Candy` object.
2. **`Candy`**: This is the name of the singleton object, representing a specific item (candy) in this case.
3. **`: VendingItem()`**: This means that `Candy` is a subclass of the sealed class `VendingItem`. It inherits from `VendingItem`, which means it has the behaviors and properties of `VendingItem`.

Why Use `object` Here?

When you use `object`, you're saying that this is a **single, unchangeable instance**. In the context of vending items, you might have different types of chips or sodas with varying properties (flavors, brands), so you'd use classes for those. But for something like `Candy`, if there's no variation (it's always the same candy), you just create one single instance using `object`.

You don't need to create multiple instances of `Candy`. It's like saying, "There's only one type of candy in this vending machine, and we don't need more than one."

Example:

```
sealed class VendingItem {
    class Chips(val flavor: String) : VendingItem()
    class Soda(val brand: String) : VendingItem()
    object Candy : VendingItem()  // Single instance of Candy
}

fun selectItem(item: VendingItem): String {
    return when (item) {
        is VendingItem.Chips -> "Selected Chips with flavor: ${item.flavor}"
        is VendingItem.Soda -> "Selected Soda of brand: ${item.brand}"
        VendingItem.Candy -> "Selected Candy"  // No need to check properties, only one
instance
    }
}
```

Key Points:

- **`object Candy`**: Only one instance of `Candy` exists, and you access it directly as `VendingItem.Candy`. There's no need to use `new` or create multiple `Candy` objects.
- **Use Case**: This is useful when you don't need variations or different instances of a subclass. For example, if all "Candy" items are the same and you don't need to track specific properties (like flavor), a singleton object makes sense.

When to Use:

You should use `object` in sealed classes when you need a single, fixed instance of a subclass, without any variation.

- Sealed class is a class that restricts class Hierarchy.
- It states- object data types that have only fixed set of possible types (subclasses).
- This is particularly useful when you want to represent a limited number of possibilities and ensure that all the cases are handled.
- Sealed class is used when the object has one of the types from limited set, but cannot have any other type.
- Enum classes represent a fixed set of values with fixed state, whereas sealed classes can represent a fixed set of classes with varying state and behavior
- We can define our child class of sealed classes either inside its body or outside as well.
- All subclasses of a sealed class must be declared in the same file where sealed class is declared.
- A sealed class is abstract by itself, and you can't initiate an object from it.
- You cannot create a non-private constructor their constructor are protected by default So a child class can call them but other class cannot.