Problem Statement: Zoo Animal Registry

You are tasked with building a **Zoo Registry System** to manage a collection of different animal objects in a zoo. The registry needs to support the following features using **Swift's object-oriented and type system**:

🎯 Your goals:

1. **Model the Animal Kingdom**:
    - Create a base class `Animal` with a `name` property.
    - Define three subclasses of `Animal`:
        - `Dog` with a `bark()` method
        - `Cat` with a `meow()` method
        - `Bird` with a `sing()` method
2. **Store a Heterogeneous Collection**:
    - Create a Swift array of type `[Any]` called `zoo` that contains:
        - At least one object of each animal subclass
        - At least one object **not** belonging to the `Animal` hierarchy (e.g., a `String` or `Int`)
3. **Loop Through the Collection**:
    - Iterate through the `zoo` array and:
        - Use the `type(of:)` function to print the **actual runtime type** of each object.
        - Use the `is` operator to check whether the object is a specific subclass (`Dog`, `Cat`, or `Bird`).
        - After confirming the type using `is`, use **safe forced casting (`as!`)** to cast and invoke the specific method of that subclass.
        - In one of the type checks (e.g., `Cat`), demonstrate the use of `as?` instead.
        - If the object is **not** an `Animal` (e.g., a `String`), print a message saying it's an unrecognized type.
4. **Safety First**:
    - Ensure that **forced casting (`as!`) is only used when it is provably safe**, such as after a preceding `is` check.
    - The program should not crash regardless of what types are in the zoo array.

Full Swift Code

C/C++

```
// MARK: - Step 1: Base class ----------------------------------------------

/// The common superclass for every animal in the zoo.
/// It stores a single property—`name`—shared by all subclasses.
class Animal {
    var name: String                      // Every animal has a readable name

    /// Designated initializer for Animal.
    /// - Parameter name: The animal's display name.
    init(name: String) {
        self.name = name
    }
}
```

C/C++

```
// MARK: - Step 2: Subclasses with unique behaviour ------------------------

/// `Dog` inherits from `Animal` and gains a `bark()` method.
class Dog: Animal {
    func bark() {                         // Dog-specific sound
        print("🐶 \(name) says: Woof!")
    }
}

/// `Cat` inherits from `Animal` and gains a `meow()` method.
class Cat: Animal {
    func meow() {                         // Cat-specific sound
        print("🐱 \(name) says: Meow!")
    }
}
```

```
/// `Bird` inherits from `Animal` and gains a `sing()` method.
class Bird: Animal {
    func sing() {                         // Bird-specific sound
        print("🐦 \(name) says: Tweet!")
    }
}
```

C/C++

```
// MARK: - Step 3: A heterogeneous collection
------------------------------

/// The zoo can hold *any* kind of object, not just `Animal`
instances.
/// Storing them as `[Any]` makes the example realistic but
requires run-time checks.
let zoo: [Any] = [
    Dog(name: "Buddy"),          // Dog
    Cat(name: "Whiskers"),       // Cat
    Bird(name: "Tweety"),        // Bird
    "Top-Secret Crate",          // A plain `String` (not an
Animal)
    Dog(name: "Shadow")          // Another Dog
]
```

C/C++

```
// MARK: - Step 4: Loop with `is` + *safe* `as!`
----------------------------

for item in zoo {

    // 1️⃣ Always start by printing the *dynamic* type.
    print("🔍 Found object of type:", type(of: item))
```

```
    // 2️⃣ --- DOG BRANCH
---------------------------------------------------
    //
    // First, *test* with `is`.  This does not cast; it only
answers
    // “Does `item` refer to a `Dog` at run-time?”
    //
    if item is Dog {
        // At this point Swift has proved that `item` *really is*
a Dog.
        // Therefore a *forced* cast using `as!` is now
guaranteed to succeed.
        let dog = item as! Dog   // ← safe because we *just*
confirmed with `is`
        dog.bark()               // Call Dog-specific behaviour
    }

    // 3️⃣ --- CAT BRANCH
---------------------------------------------------
    //
    // Here we combine test *and* conditional down-cast in one
step using `as?`.
    // If the cast fails it returns `nil`, keeping the program
safe.
    //
    else if let cat = item as? Cat {
        cat.meow()
    }

    // 4️⃣ --- BIRD BRANCH
--------------------------------------------------
    else if item is Bird {        // Another example of `is`
        // We can also do a two-step approach: check first, then
cast forced.
        // Demonstrates `as!` safely after an `is` gate.
```

```
        let bird = item as! Bird
        bird.sing()
    }

    // 5️⃣ --- FALLBACK
-----------------------------------------------------
    //
    // Anything reaching here is *not* a Dog, Cat, or Bird, so
treat as exotic.
    //
    else {
        print("⚠️  Item is not a recognised Animal:", item)
    }
}
```

🧠 Detailed Topic-by-Topic Explanation

#	Concept	Where & Why
1	**`is` Operator**	Lines like `if item is Dog` perform **type inspection** without casting. It's ultra-cheap and never crashes because it only asks a yes/no question.
2	**Forced cast `as!`**	Immediately *after* an `is` check, we know the cast **must succeed**. Doing `let dog = item as! Dog` is therefore 100 % safe (the force-cast cannot fail). This pattern ("check-then-cast") is the only recommended place to use `as!`.
3	**Optional cast `as?`**	In the Cat branch we show the traditional safe cast: `if let cat = item as? Cat { … }`. If the cast fails, `cat` is `nil` and the branch is skipped.
4	**`type(of:)` Function**	Prints the **dynamic (runtime) type** for debugging or logging. Handy when dealing with `[Any]`.

5	**Polymorphism & Inheritance**	Even though we store items as `Any`, once we down-cast them we regain full access to subclass-specific APIs (e.g., `bark()`, `meow()`).
6	**Robust Error Handling**	The final `else` keeps the program stable if the array contains *non-Animal* objects (here a `String`).

✅ Key Take-away

Combining `is` for **verification** with `as!` for **guaranteed success** is a safe and sometimes more concise alternative to `as? + if let`.
Never use `as!` without an upfront check (or other absolute guarantee), because a failed forced cast will crash the app.
