Kotlin

Here's an **overview of the Kotlin topics covered**, with space for adding links for detailed explanations of each topic:

Kotlin Overview

This document covers a wide range of Kotlin features, from basic syntax to advanced concepts like lambdas, higher-order functions, collections, nullability, and design patterns.

---

1. Kotlin Basics

- **Variable Declarations**: `var`, `val`, and `const`.
- **Conditionals**:
  - `if`, `else if`, and `else`.
  - `when` expressions for cleaner branching logic.
- **Functions as First-Class Citizens**: Assign functions to variables or pass them as arguments.
- **Lambda Expressions**: Concise syntax for anonymous functions.

Kotlin Basics

---

2. Kotlin Collections

- **Types**:
  - Immutable collections: `List`, `Set`, `Map`.
  - Mutable collections: `MutableList`, `MutableSet`, `MutableMap`.
- **Common Operations**:
  - `forEach`: Iterates over elements.
  - `map`: Transforms a collection.
  - `filter`: Filters elements based on a condition.
  - `groupBy`: Converts a list into a map based on a property.
  - `fold`: Reduces a collection to a single value.

Example:

```
val cookies = listOf(Cookie("Choco", true, 2.5))
val softBaked = cookies.filter { it.softBaked }
```

Kotlin Collections

---

3. Kotlin Nullability

- Kotlin enforces **null safety** with nullable (`?`) and non-nullable types.
- **Operators**:
  - Safe Call Operator (`?.`): Access properties/methods safely.
  - Not-Null Assertion (`!!`): Force access but risky.
  - Elvis Operator (`?:`): Provide default values for `null`.
- Examples:
  - Safe Call: `favoriteActor?.length`.
  - Elvis: `val result = favoriteActor?.length ?: 0`.

Kotlin Nullability

---

4. Lambdas and Higher-Order Functions

- **Lambda Functions**: Compact syntax for anonymous functions.
- **Higher-Order Functions (HoFs)**:
  - Functions that accept or return other functions.
- Examples:

```
val sum = { x: Int, y: Int -> x + y }
val students = listOf(Student(20, 2025))
val oldest = students.maxByOrNull { it.age }
```

Kotlin Lambda

---

5. Boxing and Unboxing

- **Boxing**: Converting primitive types (`int`, `double`) into their wrapper classes (`Integer`, `Double`).
- **Unboxing**: Converting wrapper classes back to primitives.

- **Use Cases**:
  - Nullable primitives.
  - Collections requiring object types.

Boxing and Unboxing

---

6. Subject-Observer Design Pattern in Kotlin

- **Observer**: Listens for updates from a `Subject`.
- **Subject**: Maintains state and notifies observers of changes.
- Example:

```
class Subject {
    fun attach(observer: Observer) { ... }
    fun notifyObservers() { ... }
}
```

Subject-Observer Design Pattern in Kotlin