Boxing and Unboxing

1. Overview of Boxing and Unboxing

Boxing and unboxing refer to the process of converting **primitive types** into **wrapper classes** and vice versa. While Kotlin itself handles this process seamlessly, understanding how boxing works is essential for interoperability with Java, where wrapper classes are extensively used.

---

2. Primitives and Wrapper Classes

Kotlin supports both primitive types (e.g., `int`, `double`, `char`) and their corresponding wrapper classes in Java. This is necessary for tasks such as working with nullable types, generic collections, and passing by reference.

Java Wrapper Classes and Their Primitives:

Wrapper Class	Primitive Type
`Integer`	`int`
`Double`	`double`
`Boolean`	`bool`
`Character`	`char`

---

3. Why Use Wrappers?

Wrapper classes provide additional functionality compared to primitive types:

1. **Allow `null` values**: Primitive types cannot hold `null`, but wrapper classes can.
2. **Collections of generics**: For example, `List<ClassType>` cannot work directly with primitives but requires their wrapper classes.
3. **Methods using generic `Object` types**: Many Java methods that accept `Object` types require wrapper classes for primitives.
4. **Pass by reference**: Wrappers enable object-like behavior for values stored in the heap instead of the stack.

---

4. Null Pointer Exception and Wrappers

One major drawback of working with wrapper classes is the risk of **NullPointerExceptions (NPEs)**. Kotlin mitigates this risk using its **null-safety** system, including nullable types (`?`), safe call operators (`?.`), and the Elvis operator (`?:`).

---

5. Basic Boxing and Unboxing in Kotlin

Kotlin handles boxing and unboxing automatically, allowing seamless interaction between primitive types and their wrappers.

Example:

```
fun main() {
    var x: Int = 10 // Primitive type
    println("Hello, world!!! ${x}") // Auto-boxed to Integer when needed
}
```

---

6. Boxing with Classes

In Kotlin, primitive types can be encapsulated in classes to demonstrate boxing behavior.

Example: Boxing with a `Simple` Class

```
class Simple(val arg: Int) {
    private var localArg: Int = arg

    public fun getLocalArg(): Int {
        return localArg
    }
}

fun main() {
    val simple1: Simple = Simple(99)
    val simple2: Simple? = null
    val simple3: Simple? = simple1 as Simple // Casting

    println("Here: ${simple1.getLocalArg()}")  // Output: 99
    println("Here: ${simple2?.getLocalArg()}") // Output: null
    println("Here: ${simple3?.getLocalArg()}") // Output: 99
}
```

7. The Elvis Operator (`?:`)

The Elvis operator provides a concise way to handle `null` values by specifying a default value to use when a nullable variable is `null`.

Example:

```
fun main() {
    var simple1: Simple = Simple(99)
    var simple2: Simple? = null

    val num1 = simple2?.getLocalArg() ?: -1
    println("num1: $num1")  // Output: -1

    val num2 = simple1.getLocalArg() ?: -1
    println("num2: $num2")  // Output: 99
}
```

---

8. Boxing and Boolean Types

Even Boolean values can be boxed and used with null safety.

Example:

```
fun main() {
    var i: Boolean = true
    var k = i ?: false // Elvis operator handles null values
    println(k)  // Output: true
}
```

---

9. Full Example of Boxing and Safe Calls

The following example demonstrates a combination of boxing, unboxing, safe calls, and the Elvis operator.

Example:

```
class Simple(val arg: Int) {
    private var localArg: Int = arg
```

```
    public fun getLocalArg(): Int {
        return localArg
    }
}

fun main() {
    var x: Int = 10
    println("Hello, world!!! $x")

    var simple1: Simple = Simple(99)
    var simple2: Simple? = null
    var simple3: Simple? = simple1 as Simple

    println("Here: ${simple1.getLocalArg()}")  // Output: 99
    println("Here: ${simple2?.getLocalArg()}") // Output: null
    println("Here: ${simple3?.getLocalArg()}") // Output: 99

    val num = simple2?.getLocalArg() ?: -1
    println("num: $num")  // Output: -1
}
```

---

10. Key Points

1. **Automatic Boxing and Unboxing**:
   - Kotlin automatically converts between primitive types and wrapper classes where necessary.
2. **Null-Safety**:
   - Kotlin's nullable types (`?`) and safe call operators (`?.`) prevent null pointer exceptions commonly associated with wrappers.
3. **Elvis Operator (`?:`)**:
   - Provides a concise way to handle nullable values by specifying a default when `null` is encountered.
4. **Wrapper Usage**:
   - Wrapper classes are crucial when working with generics, collections, or Java APIs requiring `Object` types.