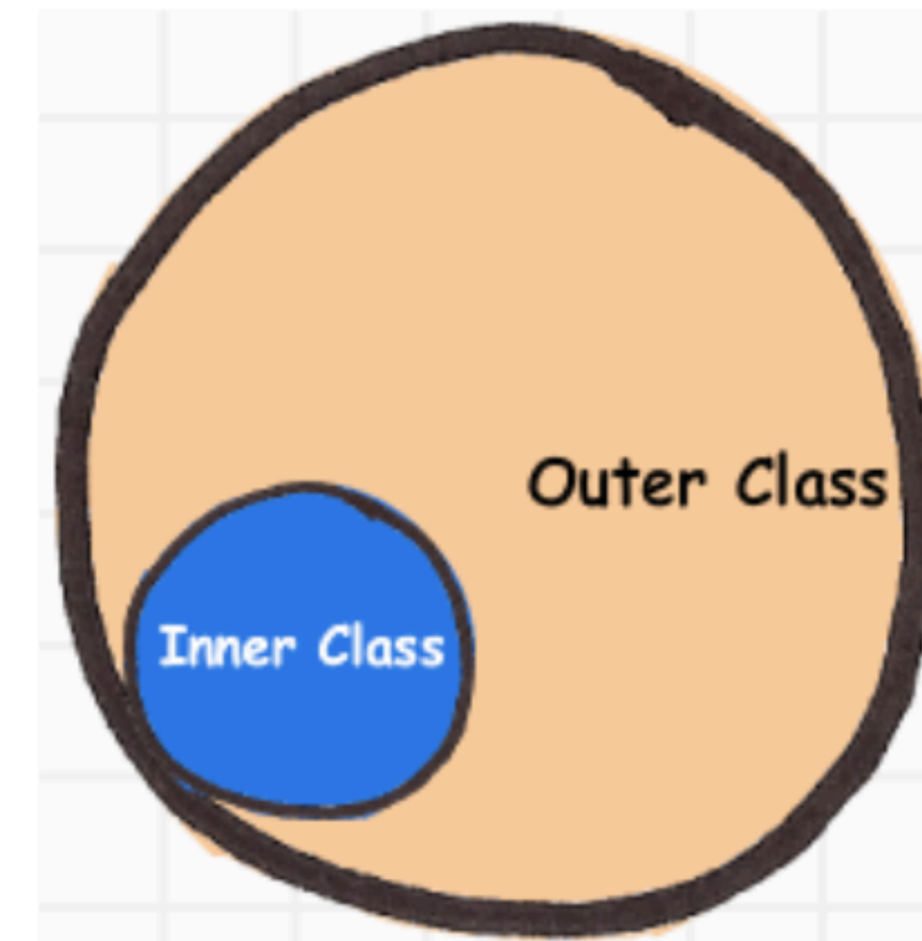
Nested Classes & Enums

Nested Classes & Enums



Classes can be nested in other classes. Enums provide an elegant notation for a limited set of constant values

Nested and Inner Classes

Classes can be nested in other classes:

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

Inner classes

A class may be marked as `inner` to be able to access members of outer class. Inner classes carry a reference to an object of an outer class:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

Anonymous inner classes

Anonymous inner class instances are created using an object expression:

```
indow.addMouseListener(object: MouseAdapter() {

   override fun mouseClicked(e: MouseEvent) { ... }

   override fun mouseEntered(e: MouseEvent) { ... }
)
```

If the object is an instance of a functional Java interface (i.e. a Java interface with a single abstract method), you can create it using a lambda expression prefixed with the type of the interface:

```
val listener = ActionListener { println("clicked") }
```

Enum Classes

The most basic usage of enum classes is implementing type-safe enums:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

Each enum constant is an object. Enum constants are separated with commas.

Initialization

Since each enum is an instance of the enum class, they can be initialized as:

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}
```

Anonymous Classes

Enum constants can also declare their own anonymous classes:

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

with their corresponding methods, as well as overriding base methods. Note that if the enum class defines any members, you need to separate the enum constant definitions from the member definitions with a semicolon, just like in Java.

Enum entries cannot contain nested types other than inner classes (deprecated in Kotlin 1.2).

Since Kotlin 1.1, it's possible to access the constants in an enum class in a generic way, using the `enumValues<T>()` and `enumValueOf<T>()` functions:

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}

printAllValues<RGB>() // prints RED, GREEN, BLUE
```

Every enum constant has properties to obtain its name and position in the enum class declaration:

```
val name: String
val ordinal: Int
```

The enum constants also implement the Comparable interface, with the natural order being the order in which they are defined in the enum class.