Development Scenario: Event Management System

Day 1: Introduction and Setup

Task 1: Set up the Kotlin development environment and write a simple Kotlin script to validate the setup.

Task 2: Experiment with Kotlin's string templates to create dynamic welcome messages.

Task 3: Define data types to represent event details such as name, date, and attendee count.

Task 4: Implement a basic user input flow to create new events using if and when statements.

```
import java.time.LocalDate
import java.time.format.DateTimeFormatter
import java.time.format.DateTimeParseException
import java.util.Scanner


fun main() {
    println("Kotlin setup is successful!")

    val userName = "Alice"
    println("Welcome, $userName! We're excited to have you.")
```

```
    data class Event(val name: String, val date: LocalDate, val attendeeCount: Int)

    val scanner = Scanner(System.`in`)

    print("Enter event name: ")
    val eventName = scanner.nextLine()

    var eventDate: LocalDate? = null
    while (eventDate == null) {
        print("Enter event date (YYYY-MM-DD): ")
        val dateString = scanner.nextLine()
        try {
            eventDate = LocalDate.parse(dateString, DateTimeFormatter.ISO_DATE)
        } catch (e: DateTimeParseException) {
            println("Invalid date format. Please use YYYY-MM-DD.")
        }
    }

    print("Enter attendee count: ")
    val attendeeCount = scanner.nextInt()

    val newEvent = Event(eventName, eventDate, attendeeCount)
    println("Event created: $newEvent")
}
```

Day 2: Functions and OOP Basics

Task 5: Design a EventManager class with methods to add and remove events.

Task 6: Create a Display interface with a method to show event details and implement it in the EventManager.

Task 7: Utilize higher-order functions to implement a simple notification system for event updates.

Task 8: Construct subclass SpecialEvent with additional features like VIP lists and premium services.

```
import java.time.LocalDate

data class Event(val name: String, val date: LocalDate, val attendeeCount: Int)
interface Display {
    fun displayEventDetails(event: Event)
}

class EventManager : Display {
    private val events = mutableListOf<Event>()

    fun addEvent(event: Event) {
        events.add(event)
    }
```

```
    fun removeEvent(eventName: String) {
        events.removeIf { it.name == eventName }
    }

    override fun displayEventDetails(event: Event) {
        println("Event: ${event.name}, Date: ${event.date}, Attendees: ${event.attendeeCount}")
    }

    fun notifyEventUpdates(notificationFunction: (Event) -> Unit) {
        events.forEach(notificationFunction)
    }
}

class SpecialEvent(
    name: String,
    date: LocalDate,
    attendeeCount: Int,
    val vipList: List<String>,
    val premiumServices: String
) : Event(name, date, attendeeCount)

fun main() {
    val eventManager = EventManager()
```

```
val event1 = Event("KotlinConf", LocalDate.of(2024, 10, 27), 500)
val event2 = Event("AndroidDev", LocalDate.of(2024, 11, 15), 300)
eventManager.addEvent(event1)
eventManager.addEvent(event2)

eventManager.notifyEventUpdates { event ->
    println("Update: ${event.name} is scheduled for ${event.date}")
}

val specialEvent = SpecialEvent(
    "VIP Gala",
    LocalDate.of(2024, 12, 10),
    100,
    listOf("Alice", "Bob"),
    "Exclusive catering and entertainment"
)
eventManager.addEvent(specialEvent)

eventManager.displayEventDetails(specialEvent)

eventManager.removeEvent("AndroidDev")

    println("\nEvents after removing AndroidDev:")
```

```
    eventManager.notifyEventUpdates { event ->
        eventManager.displayEventDetails(event)
    }
}
```

Day 3: Interfaces, Encapsulation, and Advanced Concepts / Collections and Generics

Task 9: Develop a Schedule class that uses interfaces to ensure that all event types can be scheduled and rescheduled.

```
import java.time.LocalDate

interface Schedule {
    fun schedule(date: LocalDate)
    fun reschedule(newDate: LocalDate)
}

data class Event(val name: String, var date: LocalDate, val attendeeCount: Int)

class EventScheduler(private var event: Event) : Schedule {
    override fun schedule(date: LocalDate) {
        event.date = date
        println("${event.name} scheduled for $date")
    }
```

```
    override fun reschedule(newDate: LocalDate) {
        println("${event.name} rescheduled from ${event.date} to $newDate")
        event.date = newDate
    }

    fun getEvent(): Event = event
}

fun main() {
    val myEvent = Event("My Conference", LocalDate.of(2024, 1, 1), 100)
    val scheduler = EventScheduler(myEvent)
    scheduler.schedule(LocalDate.of(2024, 2, 15))
    scheduler.reschedule(LocalDate.of(2024, 3, 20))
    println(scheduler.getEvent())
}
```

Task 10: Secure the event data with proper encapsulation and visibility modifiers.

```
import java.time.LocalDate

class EncapsulatedEvent(val name: String, private var _date: LocalDate, val attendeeCount: Int) {
    var date: LocalDate
        get() = _date
        private set(value) {
```

```
            _date = value
        }
    fun updateDate(newDate: LocalDate) {
        date = newDate
    }
    fun getDate(): LocalDate = date
}

fun main() {
    val myEvent = EncapsulatedEvent("My Encapsulated Event", LocalDate.of(2024, 1, 1), 100)
    myEvent._date = LocalDate.of(2025,1,1)
    myEvent.updateDate(LocalDate.of(2024, 2, 1))
    println(myEvent.getDate())
}
```

Task 11: Manage a collection of events allowing filtering by date or type using Kotlin's powerful collection operations.

```
import java.time.LocalDate

data class Event(val name: String, val date: LocalDate, val type: String)

fun main() {
    val events = listOf(
        Event("KotlinConf", LocalDate.of(2024, 10, 27), "Conference"),
```

```
        Event("AndroidDev", LocalDate.of(2024, 11, 15), "Meetup"),
        Event("WebSummit", LocalDate.of(2024, 10, 10), "Conference"),
        Event("MobileConf", LocalDate.of(2024, 11, 20), "Conference")
    )

    val eventsInOctober = events.filter { it.date.monthValue == 10 }
    println("Events in October: $eventsInOctober")

        val conferences = events.filter { it.type == "Conference" }
    println("Conferences: $conferences")


    val kotlinConf = events.find{it.name == "KotlinConf"}
    println("Kotlin Conf: $kotlinConf")
}
```

Task 12: Use generics to create a flexible DataManager class capable of handling different data types, including attendees and events.

```
class DataManager<T> {
    private val data = mutableListOf<T>()

    fun add(item: T) {
```

```
        data.add(item)
    }

    fun getAll(): List<T> = data

    fun find(predicate: (T) -> Boolean): T? = data.find(predicate)

    fun filter(predicate: (T) -> Boolean): List<T> = data.filter(predicate)
}

data class Attendee(val name: String, val email: String)
data class Event(val name: String, val date: LocalDate, val type: String)

fun main() {
    val eventManager = DataManager<Event>()
    eventManager.add(Event("KotlinConf", LocalDate.of(2024, 10, 27), "Conference"))
    eventManager.add(Event("AndroidDev", LocalDate.of(2024, 11, 15), "Meetup"))

    val attendeeManager = DataManager<Attendee>()
    attendeeManager.add(Attendee("Alice", "alice@example.com"))
    attendeeManager.add(Attendee("Bob", "bob@example.com"))

    println(eventManager.getAll())
    println(attendeeManager.getAll())
```

```
    val alice = attendeeManager.find { it.name == "Alice"}
    println(alice)

    val conferences = eventManager.filter{it.type == "Conference"}
    println(conferences)
}
```

Day 4: Null Safety and Exception Handling/Advanced Features (Extensions and Coroutines)

Task 1: Ensure that the system gracefully handles null references when retrieving event data.

```
import java.time.LocalDate

data class Event(val name: String, val date: LocalDate?, val category: String? = null)

fun getEventName(event: Event?): String {
    return event?.name ?: "Unknown Event"
}
```

```
fun getEventCategory(event: Event?): String? = event?.category

fun main() {
    val event1 = Event("KotlinConf", LocalDate.of(2024, 10, 27), "Conference")
    val event2: Event? = null

    println("Event 1 Name: ${getEventName(event1)}")
    println("Event 2 Name: ${getEventName(event2)}")

    println("Event 1 Category: ${getEventCategory(event1)}")
    println("Event 2 Category: ${getEventCategory(event2)}")
}
```

Task 2: Implement try-catch blocks to handle parsing errors when reading event dates and times.

```
import java.time.LocalDate
import java.time.format.DateTimeFormatter
import java.time.format.DateTimeParseException

fun parseDate(dateString: String): LocalDate? {
    return try {
        LocalDate.parse(dateString, DateTimeFormatter.ISO_DATE)
    } catch (e: DateTimeParseException) {
```

```
        println("Error parsing date: ${e.message}")
        null
    }
}

fun main() {
    val date1 = parseDate("2024-12-25")
    val date2 = parseDate("invalid-date")

    println("Date 1: $date1")
    println("Date 2: $date2")
}
```

Task 1: Write extension functions for the Event class to add features like tagging and categorization.

```
import java.time.LocalDate

data class Event(val name: String, val date: LocalDate, var tags: MutableList<String> = mutableListOf())

fun Event.addTag(tag: String): Event {
    this.tags.add(tag)
    return this
}

fun Event.categorize(category: String): Event = this.copy(name = "$name ($category)")
```

```
fun main() {
    val event = Event("KotlinConf", LocalDate.of(2024, 10, 27))
    val taggedEvent = event.addTag("Programming").addTag("Mobile")
    val categorizedEvent = taggedEvent.categorize("Conference")

    println(categorizedEvent)
    println(event)
}
```

Task 2: Introduce coroutines to concurrently handle event bookings and cancellations.

```
import kotlinx.coroutines.*
import java.time.LocalDate
import kotlin.system.measureTimeMillis

data class Event(val name: String, val date: LocalDate)

suspend fun bookEvent(event: Event) {
    delay(1000)
    println("Event ${event.name} booked.")
```

```
}

suspend fun cancelEvent(event: Event) {
    delay(500)
    println("Event ${event.name} cancelled.")
}

fun main() = runBlocking {
    val event = Event("KotlinConf", LocalDate.of(2024, 10, 27))

    val time = measureTimeMillis {
        val bookingJob = launch { bookEvent(event) }
        val cancellationJob = launch { cancelEvent(event) }

        joinAll(bookingJob, cancellationJob)
    }

    println("Time taken: $time ms")


    val timeSequential = measureTimeMillis {
        bookEvent(event)
        cancelEvent(event)
    }
```

```
    println("Time sequential: $timeSequential ms")
}
```