# Assignment 1

## 1. Explain Array and Lists in Kotlin.

In Kotlin, both Array and List are fundamental collection types used to store groups of elements, but they serve different purposes and have distinct characteristics regarding size and mutability.

### Array

An Array in Kotlin is a collection with a **fixed size** and **mutable elements**. Once an array is created, you cannot change its size by adding or removing elements. However, you can modify the value of a component at a specific index.

Arrays are created using functions like arrayOf(), or for primitive types, specialised versions such as intArrayOf() and doubleArrayOf(). These primitive arrays offer better performance as they avoid the overhead of "boxing" primitive values into objects.

**Key Characteristics:**

- **Fixed Size:** The size is defined at initialization and cannot be altered.
- **Mutable Elements:** You can change the value at any index using myArray[index] = newValue.
- **Performance:** Excellent for storing and accessing a known number of elements, especially primitives.

**Example:**

```kotlin
// An array of 3 integers
val numbers: Array<Int> = arrayOf(10, 20, 30)
numbers[1] = 25 // OK: Modifying an element
// numbers.add(40) // Compilation Error: Size is fixed
```

### List

A List is an ordered collection of elements. Unlike arrays, Kotlin's collection framework provides two distinct interfaces for lists, clearly separating read-only and mutable collections.

1. **List<T> (Read-only):** Created with listOf(), this is an immutable collection. You cannot add, remove, or modify its elements after creation. This ensures data integrity and is preferred for creating safe, predictable APIs.

2. **MutableList<T> (Mutable):** Created with mutableListOf(), this collection is **dynamic in size**. It provides functions to add (.add()), remove (.remove()), and modify elements, allowing the list to grow and shrink.

**Key Characteristics:**

- **Dynamic Size:** MutableList can change its size. List is fixed.
- **Immutability Control:** The type system clearly separates read-only (List) from mutable (MutableList) behavior.
- **Rich API:** Lists benefit from a vast library of powerful extension functions like map, filter, forEach, etc.

**Example:**

```
// A read-only list
val immutableShapes = listOf("Circle", "Square")
// immutableShapes.add("Triangle") // Compilation Error

// A mutable list
val mutableShapes = mutableListOf("Circle", "Square")
mutableShapes.add("Triangle") // OK: Size changes
mutableShapes[0] = "Sphere"   // OK: Modifying an element
```

### When to Use Which?

- Use an **Array** when you know the exact size of the collection beforehand and require maximum performance, particularly with primitive types.
- Use a **List** for most general-purpose cases. Default to the read-only List for safety and use MutableList only when you explicitly need to modify the collection's contents or size after its creation.

## 2. Explain Loops in Kotlin with break, continue and return.

Loops in Kotlin are fundamental constructs for executing a block of code repeatedly. The most common types are for loops, which iterate over sequences (like ranges or collections), and while loops, which run as long as a condition is true. To manage the execution flow within these loops, Kotlin provides three powerful keywords: break, continue, and return.

### 1. break

The **break** keyword terminates the nearest enclosing loop immediately. Program execution resumes at the first statement following the loop. It is commonly used to stop a loop prematurely when a specific condition is met, such as finding a target value in a search.

**Example:**

```
for (i in 1..10) {
    if (i == 5) {
        break // Exits the loop when i is 5
    }
    print("$i ") // Output: 1 2 3 4
}
println("Loop finished.")
```

### 2. `continue`

The **`continue`** keyword skips the remainder of the current iteration and proceeds to the next one. Any code within the loop body after the `continue` statement is ignored for that single iteration. This is useful for bypassing specific elements without stopping the entire loop.

**Example:**

```
for (i in 1..5) {
    if (i == 3) {
        continue // Skips the print statement for this iteration
    }
    print("$i ") // Output: 1 2 4 5
}
println("Loop finished.")
```

### 3. `return`

The **`return`** keyword exits the nearest enclosing *function*, not just the loop. When a `return` is encountered inside a loop, the function immediately terminates and passes control back to its caller. This is a crucial difference: `break` exits the loop, but `return` exits the entire function.

**Example:**

```
fun findNumber(numbers: List<Int>) {
    for (num in numbers) {
        if (num == 42) {
            println("Found 42!")
            return // Exits the entire findNumber function
        }
    }
    println("42 was not in the list.")
}
```

In summary, `break` stops a loop, `continue` skips an iteration, and `return` exits the function.

## 3. Explain Inheritance in Kotlin with an example

Inheritance is a core principle of Object-Oriented Programming (OOP) that allows a class (called a subclass or child) to inherit properties and methods from another class (superclass or parent).

This mechanism promotes code reusability and establishes an "is-a" relationship, for example, a Dog "is-a" type of Animal.

In Kotlin, classes and their methods are final by default, which means they cannot be inherited or overridden. To enable inheritance, the superclass and any methods you intend to override must be explicitly marked with the open keyword.

Let's look at an example. First, we define a base Animal class:

```
// Parent/Superclass must be 'open' to be inherited from.
open class Animal(val name: String) {

    // Method must be 'open' to be overridden.
    open fun makeSound() {
        println("The animal makes a generic sound.")
    }
}
```

Here, the Animal class is open, allowing other classes to inherit from it. Its makeSound() method is also open, permitting subclasses to provide a more specific implementation.

Next, a subclass Dog can inherit from Animal using the colon (:) syntax. The Dog class must call the constructor of its parent class. To change the inherited behavior, the makeSound() method is redefined using the override keyword. The subclass can also have its own unique members, like the wagTail() method.

```
// Child/Subclass inheriting from Animal
class Dog(name: String) : Animal(name) {

    // Overriding the parent's method
    override fun makeSound() {
        println("$name says: Woof!")
    }

    fun wagTail() {
        println("$name is wagging its tail.")
    }
}
```

When an object of the Dog class is created, it has access to members from both its own class and its parent class:

```
fun main() {
    val myDog = Dog("Buddy")
    println("Name: ${myDog.name}") // Inherited property from Animal
    myDog.makeSound()          // Overridden method from Dog
    myDog.wagTail()            // Method unique to Dog
}
```

This demonstrates how the `Dog` object combines inherited functionality from `Animal` with its own specialized features, resulting in a clear and hierarchical code structure.

## 4. Write a note on JSON. Difference between JSON and XML.

### A Note on JSON

JSON, an acronym for JavaScript Object Notation, is a lightweight, text-based format for data interchange. Derived from a subset of JavaScript, it is designed to be both human-readable and easy for machines to parse and generate. Despite its origin, JSON is language-independent, with parsers available for virtually all modern programming languages.

Its structure is built on two fundamental concepts:

1. **Objects**: An unordered collection of key-value pairs, enclosed in curly braces `{}`. Keys must be strings, and values can be a string, number, boolean (`$true$/$false$`), `$null$`, another object, or an array.
   - Example: `{"name": "Alex", "age": 30, "isStudent": false}`
2. **Arrays**: An ordered list of values, enclosed in square brackets `[]`.
   - Example: `["apple", "banana", "cherry"]`

Due to its simplicity and efficiency, JSON has become the de facto standard for exchanging data between a server and a web application, particularly in RESTful APIs.

### Difference Between JSON and XML

While both JSON and XML (eXtensible Markup Language) are used to structure and transport data, they have fundamental differences:

- **Syntax and Verbosity**: XML is significantly more verbose, using a tag-based structure with opening and closing tags for every data element (e.g., `<name>Alex</name>`). JSON is more compact, using key-value pairs (e.g., `"name": "Alex"`), which reduces file size and network overhead.
- **Parsing**: JSON is generally faster and easier for machines to parse, especially in web environments, as it maps directly to native JavaScript objects. XML parsing requires a more complex Document Object Model (DOM) parser, which can be slower and more memory-intensive.
- **Data Types**: JSON has built-in support for data types like numbers, strings, booleans, and arrays. In XML, all data is treated as a string by default, and data types must be explicitly defined using a schema (like XSD).
- **Arrays**: Representing arrays is a native concept in JSON. In XML, arrays are not a core feature and must be implemented by convention, often by having multiple elements with the same tag name under a parent element.

In summary, JSON is preferred for data interchange in web applications due to its conciseness and ease of parsing, while XML's extensibility and support for schemas, namespaces, and comments make it suitable for complex document markup and enterprise systems.

# Assignment 2

## 1. How can you store data from an Android application in SQLite/MySQL? Explain with an example.

In Android development, SQLite and MySQL serve distinct purposes. SQLite is an on-device database for local storage, while MySQL is a server-side database used for centralized, multi-user data.

### 1. Storing Data in SQLite (On-Device)

SQLite is built into the Android OS, making it ideal for storing private application data like user preferences, offline content, or a shopping cart. The process involves using Android's built-in `SQLiteOpenHelper` class.

**Process:**

1. **Define a Schema:** Create a "Contract" class that defines your table name and column names.
2. **Create a Helper Class:** Subclass `SQLiteOpenHelper`. In its `onCreate()` method, you execute the `CREATE TABLE` SQL statement.
3. **Insert Data:** Get a writable instance of your database and use a `ContentValues` object to map column names to the data you want to store.

**Example: Storing a username**

Let's assume you have a `DbHelper` class extending `SQLiteOpenHelper`.

Java
```
// Get a writable database instance
SQLiteDatabase db = dbHelper.getWritableDatabase();

// Create a map of values, where columns are the keys
ContentValues values = new ContentValues();
values.put(UserContract.UserEntry.COLUMN_NAME, "John Doe");

// Insert the new row, returning the primary key of the new row
long newRowId = db.insert(UserContract.UserEntry.TABLE_NAME, null, values);
```

### 2. Storing Data in MySQL (Server-Side)

You **never** connect an Android app directly to a MySQL database. It's insecure and impractical. Instead, the app communicates with a web service (API) that acts as a middleman.

**Process:**

1. **Backend API:** You create a web server with a backend script (e.g., using PHP, Node.js, or Python). This script contains the logic to connect to your MySQL database.
2. **Android App:** The app uses a networking library (like Retrofit or Volley) to send an HTTP request (typically a POST request) to your server's API endpoint.
3. **Data Insertion:** The server-side script receives the data from the app, validates it, and then executes an SQL `INSERT` query on the MySQL database.

**Example Flow:**

1. **Android App:** Sends a JSON object `{"name": "Jane Doe"}` to your API endpoint, `https://api.yourdomain.com/adduser`.

**Server (e.g., PHP script):**
```
<?php
// Connect to MySQL database (credentials not shown)
$dbConnection = ...;

// Get data from the app's request
$name = $_POST['name'];

// Prepare and execute the SQL statement
$stmt = $dbConnection->prepare("INSERT INTO users (name) VALUES (?)");
$stmt->bind_param("s", $name);
$stmt->execute();
?>
```

## 2. How can you access the current location and the camera of the user's device? Explain it.

In modern mobile development with Kotlin for Android, you cannot access a user's location or camera without their explicit, runtime permission. This is a fundamental security and privacy feature of the Android operating system. The process involves requesting permissions and then using specific APIs to access the hardware.

### Accessing the User's Location

1. **Declare Permissions:** First, you must declare the necessary permissions in your `AndroidManifest.xml` file. For location, you'll add either `ACCESS_COARSE_LOCATION` (for approximate location) or `ACCESS_FINE_LOCATION` (for precise location).

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

2. **Request Runtime Permission:** Starting from Android 6.0 (API level 23), you must request these "dangerous" permissions from the user while the app is running. You check if the permission has already been granted. If not, you launch a system dialog to ask the user.
3. **Fetch Location Data:** Once permission is granted, you use the **Fused Location Provider API** from Google Play Services. This is the recommended approach as it's power-efficient and simple. You can get the last known location or request continuous location updates.

```
// Conceptual code after permission is granted
private lateinit var fusedLocationClient: FusedLocationProviderClient
fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)

fusedLocationClient.lastLocation
    .addOnSuccessListener { location : Location? ->
        // Got last known location. In some rare situations this can be null.
    }
```

## Accessing the User's Camera

1. **Declare Permission & Feature:** Add the `CAMERA` permission and declare the camera feature in your `AndroidManifest.xml`.

   ```
   <uses-permission android:name="android.permission.CAMERA" />
   <uses-feature android:name="android.hardware.camera" />
   ```

2. **Request Runtime Permission:** Just like with location, you must request the `CAMERA` permission at runtime if it hasn't been granted.
3. **Use the Camera:**
   - **Intent Method (Simple):** The easiest way is to use an `Intent` with `MediaStore.ACTION_IMAGE_CAPTURE`. This delegates the task to the user's default camera app. Your app receives the resulting image back. This is secure and requires less code.
   - **CameraX API (Advanced):** For building a custom camera experience within your app, you should use the **CameraX** Jetpack library. It provides a robust, easy-to-use API surface that simplifies working with the underlying `Camera2` framework, handling lifecycle and device-specific complexities for you.