در اینجا به چند مفهوم باید دقت شود ۱ مفهوم Array که اندازه ثابتی دارد و عناصر در آن ه ترتیب در حافظه قرار می‌گیرند و امکان تغییر در آنها نیست اما مفهوم دیگر کالکشن‌ها هستند که اندازه ثابتی ندارند و شامل لیست و ست و مپ هستند

نکته دیگر اینکه رایه‌ها در کاتلین یک کلاس هستند

نکته دیگر اینکه ۱ArrayList یک typealias از java.util.ArrayList زبان جاوا هستند که در ر به تفاوت‌های این دو نیز پرداخته می‌شود

## Array:

You may consider an array as a collection of elements of one type. All elements are stored in the memory sequentially.

The collection provides one name for all elements. Once an array is created, it cannot be changed. However, you can modify a stored element at any time.

Kotlin can handle many types of arrays: IntArray, LongArray, DoubleArray, FloatArray, CharArray, ShortArray, ByteArray, BooleanArray. Each array stores elements of the corresponding type (Int, Long, Double, and so on). Note that Kotlin doesn't have a default StringArray type. You can store Strings in arrays.

To create an array of a specified type, we need to invoke a special function and pass all elements to store them together:

- intArrayOf creates IntArray;
- charArrayOf creates CharArray;
- doubleArrayOf creates DoubleArray;

```
val numbers = intArrayOf(1, 2, 3, 4, 5) // It stores 5 elements of the Int type
println(numbers.joinToString()) // 1, 2, 3, 4, 5

val characters = charArrayOf('K', 't', 'l') // It stores 3 elements of the Char type
println(characters.joinToString()) // K, t, l

val doubles = doubleArrayOf(1.25, 0.17, 0.4) // It stores 3 elements of the
Double type
println(doubles.joinToString()) // 1.25, 0.17, 0.4
```

## Comparing arrays:

To compare two arrays, invoke the contentEquals() function of an array and pass another array as the argument. This function returns true when two arrays contain the same elements in the same order. Otherwise, it returns false:

```
val numbers1 = intArrayOf(1, 2, 3, 4)
val numbers2 = intArrayOf(1, 2, 3, 4)
val numbers3 = intArrayOf(1, 2, 3)

println(numbers1.contentEquals(numbers2)) // true
println(numbers1.contentEquals(numbers3)) // false
```

Beware, the operators == and != do not compare the contents of arrays, they compare only the variables that point to the same object:

```
val simpleArray = intArrayOf(1, 2, 3, 4)
val similarArray = intArrayOf(1, 2, 3, 4)

println(simpleArray == simpleArray)  // true, simpleArray points to the same object
println(simpleArray == similarArray) // false, similarArray points to another object
```

### What is a collection?

Collections are containers that support various ways to store and organize different objects and make them easily accessible. A collection usually contains a number of objects (this number may be zero) of the same type. Objects in a collection are called elements or items. Collections are an implementation of abstract data structures that can support the following operations:

- retrieving an element;
- removing an element;
- changing or replacing an element;
- adding a new element.

However, it is important to note that operations like adding, removing, and changing elements only apply to mutable collections. Let's try to understand what this means.

Mutability:

Immutable collections cannot be changed. That is, they only allow those operations that do not change the elements, such as accessing an element. Immutability can be helpful when you want to store items together without allowing them to be modified in the future.

Mutable collections also let you access the elements, but in addition, they allow operations that change the content of a collection by adding, removing, or updating the stored items.

Collections in Kotlin :

Kotlin Standard Library provides the implementation for the basic types of collections: list, set, and map. All three exist in mutable and immutable variations.

List stores elements in a specified order and provides indexed access to them.

Set stores unique elements whose order is generally undefined.

Map stores key-value pairs (entries); keys are unique, but different keys can be paired with equal values.

## Arrays Vs ArrayLists:

An Array (System.Array) is fixed in size once it is allocated. You can't add items to it or remove items from it. Also, all the elements must be the same type. As a result, it is type safe, and is also the most efficient of the three, both in terms of memory and performance. Also, System.Array supports multiple dimensions (i.e. it has a Rank property) while List and ArrayList do not (although you can create a List of Lists or an ArrayList of ArrayLists, if you want to).

An ArrayList is a flexible array which contains a list of objects. You can add and remove items from it and it automatically deals with allocating space. If you store value types in it, they are boxed and unboxed, which can be a bit inefficient. Also, it is not type-safe.

A List<> leverages generics; it is essentially a type-safe version of ArrayList. This means there is no boxing or unboxing (which improves performance) and if you attempt to add an item of the wrong type it'll generate a compile-time error.

## توضیح فرایند boxing و Unboxing در arrayو arrayList:

مفهوم **boxing** و **unboxing** در زمینه‌ی آرایه‌ها (Arrays) و آرایه‌های لیستی (ArrayLists) به فرآیند تبدیل بین نوع‌های اولیه (مثل `Int`, `Double`) و نوع‌های مرجع (مثل `Integer`, `Double`) اشاره دارد. در ادامه، این مفاهیم را در زمینه‌ی آرایه‌ها و آرایه‌های لیستی توضیح می‌دهیم:

### ۱. Boxing ###

**Boxing** به فرآیند تبدیل یک نوع اولیه به نوع مرجع مربوط می‌شود. زمانی که شما یک نوع اولیه (مانند `Int` یا `Double`) را در یک آرایه یا آرایه‌ی لیستی که فقط می‌تواند اشیاء (Objects) را ذخیره کند، قرار می‌دهید، آن مقدار به‌صورت یک شیء بسته‌بندی (boxed) می‌شود.

**در آرایه‌ها**:

- آرایه‌ها در زبان‌هایی مانند جاوا یا کاتلین به طور مستقیم از نوع‌های اولیه پشتیبانی می‌کنند. بنابراین، نیازی به boxing برای ذخیره نوع‌های اولیه در آرایه‌ها نیست.

``` kotlin
```

val intArray: IntArray = intArrayOf بدون // (3 ,2 ,1)boxing

```

**در آرایه‌های لیستی**:

- اگر شما بخواهید یک نوع اولیه را در یک `ArrayList` (یا `ArrayList` در جاوا) ذخیره کنید، آن مقدار به‌صورت یک شیء بسته‌بندی می‌شود. به عنوان مثال، اگر بخواهید یک عدد صحیح (Int) را در یک `ArrayList` قرار دهید، آن عدد به نوع مرجع (مثل `Integer`) تبدیل می‌شود.

```kotlin
val arrayList: ArrayList<Int> = arrayListOf(1, 2, 3) // boxingدر پس‌زمینه
```

### ۲. Unboxing

**Unboxing** به فرآیند تبدیل یک نوع مرجع به نوع اولیه مربوط می‌شود. وقتی که شما یک شیء (مثل یک boxed Integer) را از یک آرایه‌ی لیستی یا هر ساختار داده‌ای که داده‌های مرجع را نگه می‌دارد، استخراج می‌کنید، آن شیء به نوع اولیه تبدیل می‌شود.

**در آرایه‌ها**:

- در آرایه‌ها، unboxing معمولاً اتفاق نمی‌افتد زیرا نوع‌ها از ابتدا به‌طور مستقیم از نوع‌های اولیه پشتیبانی می‌کنند.

**در آرایه‌های لیستی**:

- وقتی شما یک عدد را از `ArrayList` استخراج می‌کنید، آن عدد به نوع اولیه (مثل `Int`) تبدیل می‌شود. این فرآیند ممکن است کمی کارایی را تحت تأثیر قرار دهد زیرا نیاز به تبدیل بین نوع‌ها وجود دارد.

```kotlin
val boxedInt: Int? = arrayList[0] // unboxingتبدیل عدد مرجع به نوع اولیه :
```

### نکات مهم
```

- **کارایی**: فرآیندهای boxing و unboxing می‌توانند به دلیل نیاز به تخصیص حافظه و تبدیل بین نوع‌ها، بر روی کارایی تأثیر بگذارند. این امر به ویژه زمانی اهمیت دارد که با تعداد زیادی از داده‌ها کار می‌کنید.

- **نوع‌های مرجع**: آرایه‌ها در کاتلین می‌توانند از نوع‌های اولیه به‌طور مستقیم استفاده کنند، در حالی که آرایه‌های لیستی به نوع‌های مرجع وابسته هستند و بنابراین boxing و unboxing را شامل می‌شوند.

### نتیجه‌گیری

در مجموع، boxing و unboxing در آرایه‌های لیستی به عنوان فرآیندهای ضروری برای تبدیل بین نوع‌های اولیه و نوع‌های مرجع عمل می‌کنند، در حالی که آرایه‌ها به دلیل پشتیبانی مستقیم از نوع‌های اولیه، نیازی به این فرآیندها ندارند. این تفاوت‌ها می‌توانند بر روی کارایی و نوع ایمنی برنامه تأثیر بگذارند.