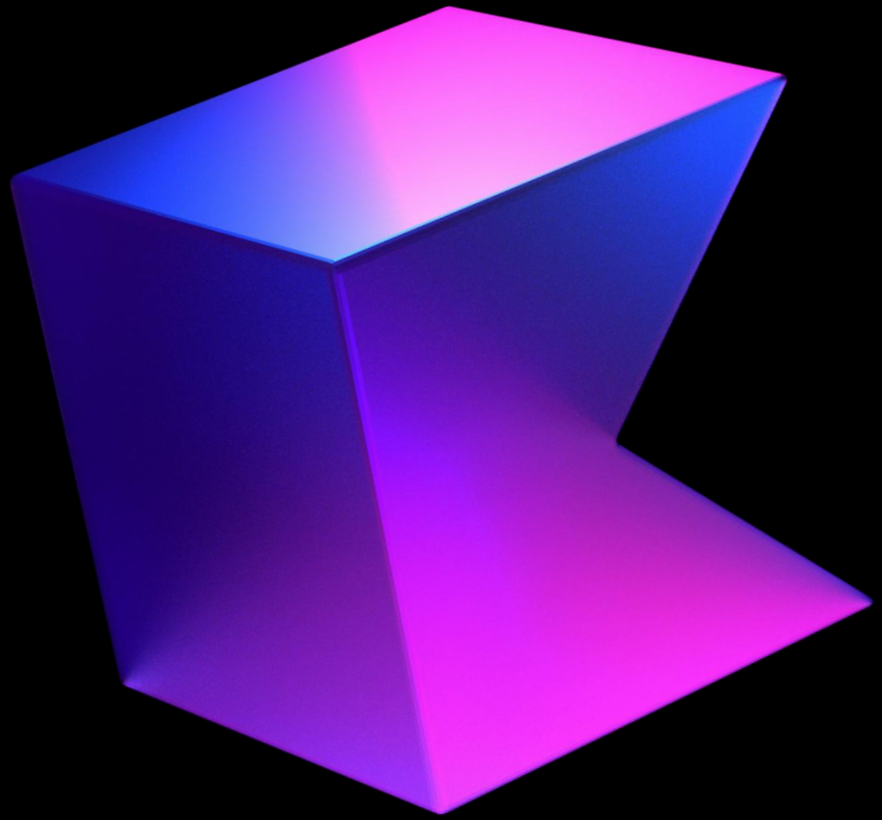
JET BRAINS Kotlin

Collections



What are they?

A collection usually contains a number of objects (this number may also be zero) of the **same type**.

Objects in a collection are called **elements** or **items**.

- **Lists** are **ordered** collections with access to elements by **indices**
  - integer numbers that reflect their position.
  - Elements can occur more than once in a list.
- **Sets** are collections of **unique elements**.
  They reflect the mathematical abstraction of “set”: a group of objects **without duplicates**.
- **Maps** (or dictionaries) are sets of key-value pairs.
  The **keys** are **unique**, and each of them maps to exactly one value,
  while the **values** can be **duplicated.**

How can they be used?

Kotlin lets you manipulate collections independently of the exact types of objects stored in them.

In other words, you add a **`String`** to a list of **`Strings`**
the same way as you would do with **`Ints`** or a user-defined class.

So, the Kotlin Standard Library offers **generic *interfaces*, *classes*,** and ***functions***
for *creating*, *populating*, and *managing* collections of any type.

	List	Set	Map (Dictionary)
Description	**Ordered** collection with access to elements by **indices** – integer numbers that reflect their position.	stores **unique elements**; their order is generally undefined.	set of **key-value** pairs. **Keys** are **unique**, and each of them maps to exactly one value. The **values** can be **duplicates**.
Order	Order is important	Undefined	Order isn't important
Accessing	Accessed via indices	Accessed via element	Accessed via keys
Uniqueness	Elements can occur more than once in a list	Unique	Unique Keys Can contain repeated values
Contains null	Yes, and can be duplicated	Can contain only one null	Can contain only one null as a key
Equality	Lists are considered equal if they have the same sizes and structurally equal elements at the same positions	sets are considered equal if they have the same size, and for each element of a set there is an equal element in the other set	Two maps are considered equal if the containing pairs are equal regardless of the pair order.

Collections Cont’d

- Collections can be
  - **Immutable** (Read Only) Collection: once it is created, its data can’t be changed
    - interface that provides operations for accessing collection elements.
    - Covariant.
    - Example: List, Set, Map
  - **Mutable** (Read – write) Collection: data contained can be changed (add – delete)
    - interface that extends the corresponding read-only interface with write operations: adding, removing, and updating its elements
    - Not covariant.
    - Example: MuatbleList, MutableSet, MutableMap

Collections under the hood: **List**

```
public interface List<out E> : Collection<E> {
        public operator fun get(index: Int): E

        public fun indexOf(element: @UnsafeVariance E): Int
        public fun lastIndexOf(element: @UnsafeVariance E): Int

        public fun subList(fromIndex: Int, toIndex: Int): List<E>

        ...
}
```

Convenient to use with []: `collection[2]`

Make a **referenced** copy:

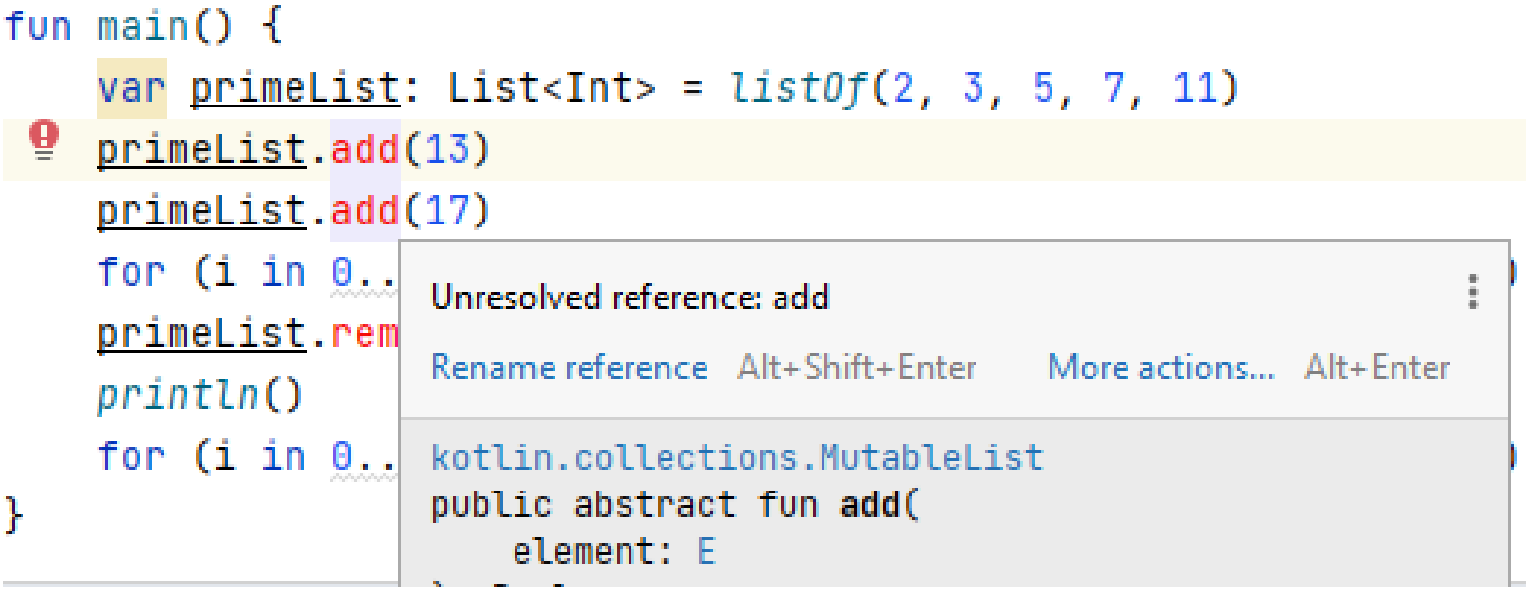
```
val list1 = mutableListOf(1, 2, 3)
val list2 = list1.subList(0, 1)
list1[0] += 1
println(list1) // [2, 2, 3]
println(list2) // [2]
```

Collections under the hood: **List**

To create a new list you can use special **builders** (by default `ArrayList`):

```
val list1 = emptyList<Int>()     // Builds the internal object EmptyList
val list2 = listOf<Int>()        // Calls emptyList()
val list3 = listOf(1, 2, 3)      // The type can be inferred

val list4 = mutableListOf<Int>()           // But better: ArrayList<Int>()
val list5 = mutableListOf(1, 2, 3)         // The type can be inferred
val list6 = buildList {
        // constructs MutableList<Int>
        add(5)
        addAll(0, listOf(1, 2, 3))
}
```

Immutable **List** example



Mutable **List** example

```
fun main() {
    val primeList: MutableList<Int> = mutableListOf(2, 3, 5, 7, 11)
    primeList.add(13)
    primeList.add(17)
    for (i in 0..primeList.size-1) print("${primeList.get(i)}  ")
    primeList.remove( element: 2)
    println()
    for (i in 0..primeList.size-1) print("${primeList.get(i)}  ")
}
```

```
CollectionsDemoKt
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
2  3  5  7  11  13  17
3  5  7  11  13  17
Process finished with exit code 0
```

List - specific operations

Operation	Function	Example
Retrieve elements by index	elementAt(), first(), last(), get(), getOrNull()	val numbers = listOf(1, 2, 3, 4) println(numbers.getOrNull(5))
Retrieve list parts	subList()	val numbers = (0..13).toList() println(numbers.subList(3, 6))
Find element positions	indexOf(number) lastIndexOf(number)	val numbers = listOf(1, 2, 3, 4, 2, 5) println(numbers.indexOf(2)) println(numbers.lastIndexOf(2))
List write operations	add(), set(), fill(), removeAt(), sort(), sortBy(), reverse(), shuffle()	val numbers = mutableListOf("one", "five") numbers.add(1, "two") numbers.addAll(2, listOf("three", "four"))

Collections under the hood: **Set**

```
public interface Set<out E> : Collection<E> {
        abstract val size: Int

        abstract fun contains(element: @UnsafeVariance E): Boolean

        abstract fun containsAll(collection: Collection<E>): Boolean

        abstract fun isEmpty(): Boolean

        abstract fun iterator(): Iterator<E>
}
```

A generic **unordered** collection of elements that **does not support duplicate elements**.
It compares objects via the **equals** method instead of checking if the objects are the *same.*

Collections under the hood: **Set**

```
class A(val primary: Int, val secondary: Int)

fun main() {
    val a = A(1,1)
    val b = A(1,2)
    val set = setOf(a, b)
    println(set) // two elements
}
```

Collections under the hood: **Set**

```
class A(val primary: Int, val secondary: Int)
class B(val primary: Int, val secondary: Int) {
    override fun hashCode(): Int = primary

    override fun equals(other: Any?) = primary == (other as? B)?.primary
}

fun main() {
    val a = B(1,1)
    val b = B(1,2)
    val set = setOf(a, b)
    println(set) // only one element
}
```

Collections under the hood: **Set**

To create a new set you can use special **builders** (by default `LinkedHashSet`):

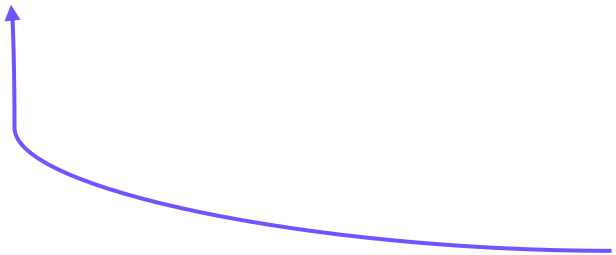
```
val set1 = emptySet<Int>()       // Builds the internal object EmptySet
val set2 = setOf<Int>()          // Calls emptySet()
val set3 = setOf(1, 2, 3)        // The type can be inferred

val set4 = mutableSetOf<Int>()             // But better: LinkedHashSet<Int>() or HashSet<Int>()
val set5 = mutableSetOf(1, 2, 3)           // The type can be inferred
val set6 = buildSet {
        // constructs MutableSet<Int>
        add(5)
        addAll(listOf(1, 2, 3))
}
```

Set - specific operations

Operation	Function	Example
Merge two collections	union()	val numbers = setOf("one", "two", "three") println(numbers union setOf("four", "five")) println(setOf("four", "five") union numbers)
Find an intersection between two collections	intersect()	val numbers = setOf("one", "two", "three") println(numbers intersect setOf("two", "one"))
Find collection elements **not** present in another collection	subtract	val numbers = setOf("one", "two", "three") println(numbers subtract setOf("four", "three"))

Collections under the hood: **Map**

```
public interface Map<K, out V> {
            public fun containsKey(key: K): Boolean

            public fun containsValue(value: @UnsafeVariance V): Boolean

            public operator fun get(key: K): V?

            public fun getOrDefault(key: K, defaultValue: @UnsafeVariance V): V

            public val entries: Set<Map.Entry<K, V>>

            …
}
```



Convenient to use in loops:

```
for ((key, value) in map.entries) { ... }
```

Collections under the hood: **Map**

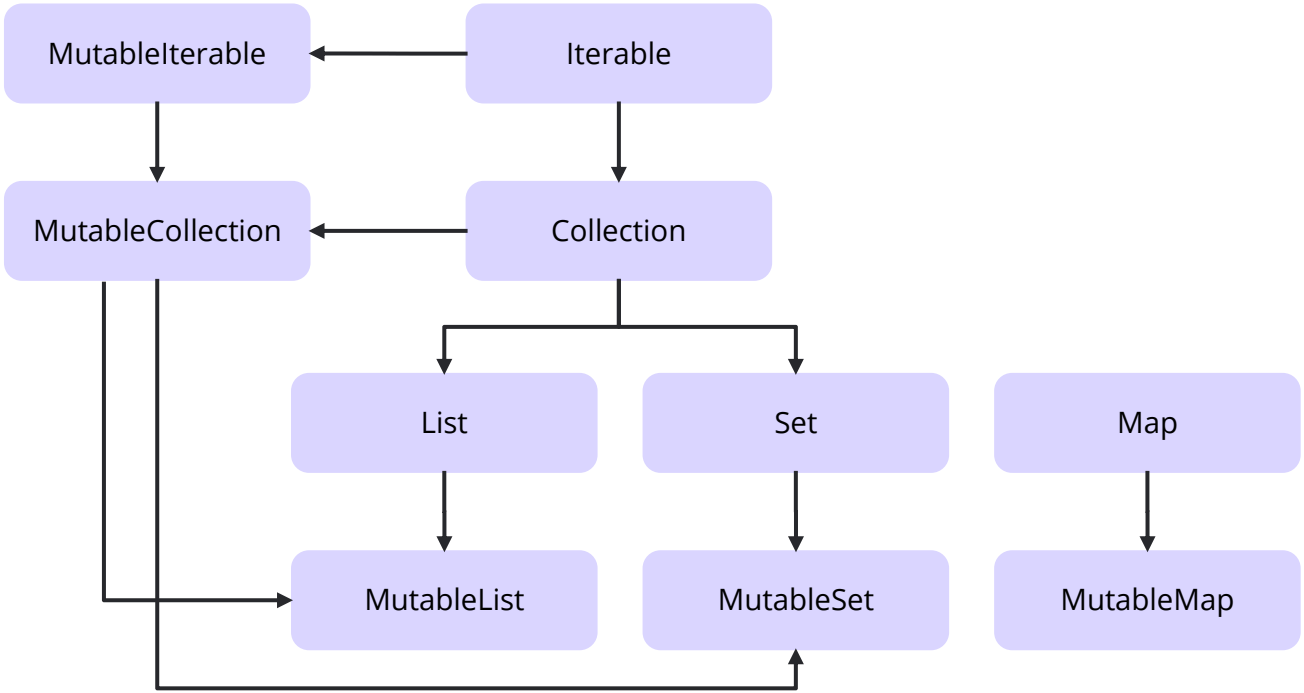
To create a new map, you can use special **builders** (by default, LinkedHashMap):

```
val map1 = emptyMap<Int, String>()          // Builds the internal object EmptyMap
val map2 = mapOf<Int, String>()             // Calls emptyMap()
val map3 = mapOf(1 to "one", 2 to "two")    // The type can be inferred

val map4 = mutableMapOf<Int, String>()      // But better: LinkedHashMap<...>() or HashMap<...>()
val map5 = mutableMapOf(1 to "one", 2 to "two") // The type can be inferred
val map6 = buildMap {
        // constructs MutableMap<Int, String>
        put(1, "one")
        putAll(mutableMapOf(2 to "two"))
    }
```

Map - specific operations

Operation	Function	Example
Retrieve keys and values	get(), getValue(), getOrDefault()	val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3) println(numbersMap.get("one")) println(numbersMap.getOrDefault("four", 10))
To perform operations on all keys or all values	keys, values	val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3) println(numbersMap.keys)
Filter	filter(), filterKeys(), filterValues()	val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11) val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
Map write operations	put(), +=, remove(), -=	val numbersMap = mutableMapOf("one" to 1, "two" to 2) numbersMap += mapOf("four" to 4, "five" to 5)

Taxonomy of collections



Iterable

All collections in Kotlin implement **`Iterable`** interface:

```
/**
* Classes that inherit from this interface can be represented as a sequence of elements that can be iterated over.
* @param T is the type of element being iterated over. The iterator is covariant in its element type.
*/
public interface Iterable<out T> {
    // Returns an iterator over the elements of this object.
   public operator fun iterator(): Iterator<T>
}
```

Iterable

All collections in Kotlin are **`Iterable`**:

```
val iterator = myIterableCollection.iterator()
while (iterator.hasNext()) {
   iterator.next()
}
```

Iterable vs MutableIterable

All collections in Kotlin are **`Iterable`**:

```
val iterator = myIterableCollection.iterator()
while (iterator.hasNext()) {
   iterator.next()
}
```

But some of them are `MutableIterable`:

```
val iterator = myMutableIterableCollection.iterator()
while (iterator.hasNext()) {
   iterator.next()
   iterator.remove() // Because it is a mutable iterator
}
```

Different kinds of collections

There are **two** kinds of collections: **`Collection`** and **`MutableCollection`**.
`Collection` implements only the **`Iterable`** interface,
while **`MutableCollection`** implements **`Collection`** and **`MutableIterable`** interfaces.

`Collection` allows you to read values and make the collection **immutable**.

`MutableCollection` allows you to **change the collection**, for example, by adding or removing elements.
In other words, it makes the collection **mutable**.

```
val readonlyCollection = listOf(1, 2, 3)
readonlyCollection.add(4) // ERROR: Unresolved reference: add

val mutableCollection = mutableListOf(1, 2, 3)
mutableCollection.add(4) // OK
```

Mutable Collection != Mutable Variable

If you create a mutable collection, you **cannot** reassign the **`val`** variable.

```
val mutableCollection = mutableListOf(1, 2, 3)
mutableCollection.add(4) // OK
mutableCollection = mutableListOf(4, 5, 6) //ERROR: Val cannot be reassigned
```

But you **can** reassign **var**.

```
var mutableCollection = mutableListOf(1, 2, 3)
mutableCollection.add(4) // OK
mutableCollection = mutableListOf(4, 5, 6) // OK
```

Today’s Challenge(s)