Maps

Understand the map data structure, its use cases, and how to put it to use in Kotlin.

WE'LL COVER THE FOLLOWING

A map is a collection of keys associated with values. This is a concept you may also know as *dictionaries* (e.g., Python) or *associative arrays* (e.g., PHP). Each key is associated with one unique value. This value can have any type. In particular, it may be a collection, such as a list or a nested map.

Creating a Map

Once again, Kotlin provides a convenient way to initialize (read-only) maps:

```
val grades = mapOf(
  "Kate" to 3.9,
  "Jake" to 3.4,
  "Susan" to 3.5
)
```

This map associated the key "Kate" to the value 3.9, "Jake" to 3.4, and so on.

> ## Pairs and Infix Functions
>
> The to function is a so-called *infix function*, a concept which we'll cover later in this course. It returns a Pair, which is a simple data structure containing exactly two values. Therefore, you can also initialize the array above using Pairs directly:
>
> ```
> val grades = mapOf(
>   Pair("Kate", 3.9),
>   Pair("Jake", 3.4),
>   Pair("Susan", 3.5)
> )
> ```
>
> Kotlin makes this initialization more convenient and readable by providing the to function.

From now on, it will be important to understand the terms map, entry, key, and value. In the example above:

- The grades variable is the *map*.
- This map contains three *entries*, and each *entry* has a key and a value.
- The first entry's *key* is "Kate", and its *value* is 3.9.

Creating a Mutable Map

As you may expect by now, Kotlin offers mutableMapOf to create a mutable map:

```
val enrollments = mutableMapOf(
  "Kate" to listOf("Maths", "Engineering"),
  "Jake" to listOf("CS", "Bioengineering", "Psychology"),
  "Susan" to listOf("Engineering", "Psychology")
)
```

By using a mutable map, you can add and remove entries from the map, as we'll cover later in this lesson.

Working with Maps

Doing useful computations on maps requires accessing and manipulating its entries.

Accessing Entries of a Map

Accessing an entry of a map is again achieved using the *indexed access operator*. But this time, we'll pass in the key of the entry we want to retrieve:

```
val enrollments = mutableMapOf(
  "Kate" to listOf("Maths", "Engineering"),
  "Jake" to listOf("CS", "Bioengineering", "Psychology"),
  "Susan" to listOf("Engineering", "Psychology")
)

val classes = enrollments["Kate"]
println(classes)   // ["Maths", "Engineering"]
```

Since you can't access the map entries by index (only by key), **maps are unordered collections**, similar to sets. Note that the value itself may again be a collection, so you could use the indexed access operator on it. You can nest collections arbitrarily deep.

> Mini Challenge
>
> Try to access Jake's third enrolled class, starting with the `enrollments` map.

If no entry with the given key exists, the indexed access operator returns `null`. To avoid nullability in your code, you can use `getOrDefault` and `getOrElse`. With `getOrDefault`, you pass in an additional default value which is used in case the key doesn't exist. With `getOrElse`, you can pass in a block of code that computes the value to use in case the key doesn't exist:

```
val enrollments = mutableMapOf(
  "Kate" to listOf("Maths", "Engineering"),
  "Jake" to listOf("CS", "Bioengineering", "Psychology"),
  "Susan" to listOf("Engineering", "Psychology")
)

val marcusClasses = enrollments.getOrDefault("Marcus", emptyList())
println(marcusClasses)   // [], not null

val simonsClasses = enrollments.getOrElse("Simon", {
    /* compute... */
    emptyList()
})
println(simonsClasses)   // [], not null
```

Both work almost the same, but `getOrElse` lets you compute more complex expressions inline, whereas, with `getOrDefault`, the desired value should be readily available.

Manipulating Entries in a Map

Assuming you have a *mutable* map, you can add, remove, and manipulate entries. To add a new entry, you use `put`:

```
enrollments.put("Marcus", listOf("Maths", "CS"))

// Can be written more idiomatically as:
enrollments["Marcus"] = listOf("Maths", "CS")
```

The `put` method accepts a key and value, similar to when you first initialize a map. The idiomatic way to write this is using square brackets, consistent with indexed access syntax.

Second, you can remove an entry using `remove` by passing the key of the entry to remove:

```
enrollments.remove("Jake")
```

Although this is by far the most common way to remove an entry, you can also pass an additional value. In that case, an entry is only removed if both its key and value match your arguments.

Third, you can overwrite the value of an existing entry (in case of a mutable map):

```
enrollments["Susan"] = listOf("CS", "Psychology")
```

This overwrites the value for the entry with key `"Susan"`.

> Note that the syntax is the same as for adding a new element. Therefore, if the given key (`"Susan"`) doesn't exist, this will add a new map entry.

If the map stored mutable lists of enrollments, you could also add or remove them from the existing classes instead of overwriting the entire list. For instance, using `enrollments["Susan"].add("Business Administration")`.

Quiz

Mutable vs Read-only Maps

1 Which of the following statements are correct?

COMPLETED 0%

1 of 4

Exercises

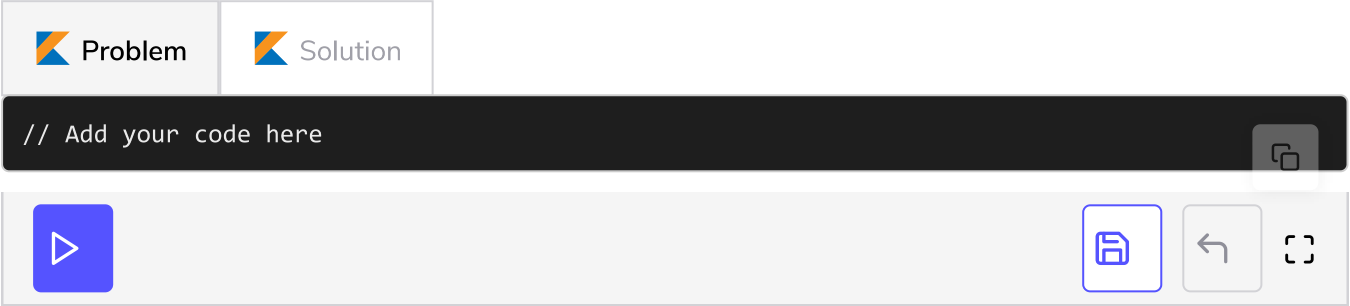
First, create a map with the past two years and current year as keys. For instance, if it's 2020, use 2018, 2019, and 2020.

Then, assign a list of things you achieved that year to each key, or assign them to whatever you were grateful for that year (maybe a holiday, gatherings with friends, a career milestone, learning a new skill, etc.).

After that, recap all main map operations using your code above:

- Make your map a mutable map
- Add a new entry for three years ago
- Remove the entry for the current year
- Make the list of achievements or things you're grateful for mutable
  - Then add one more element to the existing list for last year

Hint: You can print each intermediate map using `println(myMap)` to sanity-check your code.



```
// Add your code here
```

Summary

Maps are a fundamental data structure that you will find throughout real-world code. Therefore, mastering maps is crucial.

- Maps contains *entries* that map *keys* to *values*.
- Basic maps are created using `mapOf` or `mutableMapOf`.
  - Again, prefer read-only maps over mutable ones.
- Kotlin provides `to` for creating map entries in readable way (`myKey to myValue`).
- You can add entries to a mutable map using `myMap[key] = value`.
  - Overwriting an existing entry works the same way (if `key` already

exists).

- You can remove entries from a mutable map using `myMap.remove(key)`.

Congratulations, you are now familiar with the most widely used collection types in Kotlin. In the next section, you will learn how to use loops for control flow.