More Kotlin Fundamentals.

▼ Generics, Objects, and Extensions.

Make a Reusable Class with Generics.

Let's say you're writing an app for an online quiz. There are often multiple types of quiz questions, such as fill-in-the-blank, or true or false. An individual quiz question can be represented by a class, with several properties.

The question text in a quiz can be represented by a string. Quiz questions also need to represent the answer. However, different question types, such as true or false, may need to represent the answer using a different data type. Let's define three different types of questions.

- **Fill-in-the-blank question**: The answer is a word represented by a `String`.
- **True or false question**: The answer is represented by a `Boolean`.
- **Math problems**: The answer for a simple arithmetic problem is represented by an `Int`.

In addition, quiz questions in our example, regardless of the type of question, will also have a difficulty rating. The difficulty rating is represented by a string with 3 possible values: `"easy"`, `"medium"`, or `"hard"`.

```
class FillInTheBlankQuestion(
    val questionText: String,
    val answer: String,
    val difficulty: String
)

class TrueOrFalseQuestion(
    val questionText: String,
    val answer: Boolean,
    val difficulty: String
)
class NumericQuestion(
```

```
    val questionText: String,
    val answer: Int,
    val difficulty: String
)
```

Do you notice any repetition? All three classes have the exact same properties: the `questionText`, `answer`, and `difficulty`. The only difference is the data type of the `answer` property.

However, using inheritance has the same problem as above. Every time you add a new type of question, you have to add an `answer` property. The only difference is the data type. It also looks strange to have a parent class `Question` that doesn't have an answer property.

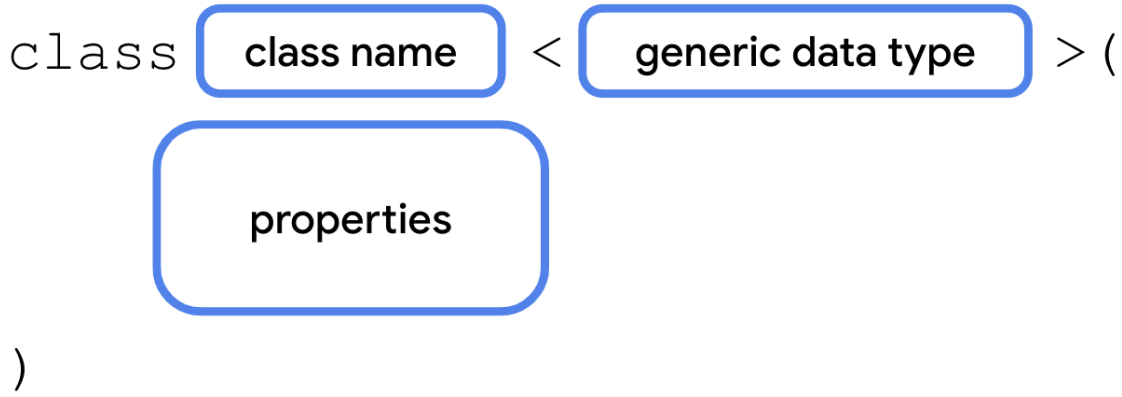
When you want a property to have differing data types, subclassing is not the answer. Instead, Kotlin provides something called *generic types* that allow you to have a single property that can have differing data types, depending on the specific use case.

What is a Generic Data Type?

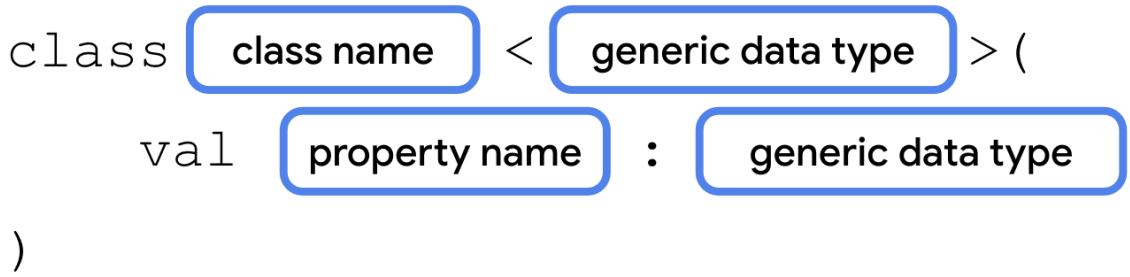
> 💡 Generics, allow a data type, such as a class, to specify an unknown placeholder data type that can be used with its properties and methods.

In the above example, instead of defining an answer property for each possible data type, you can create a single class to represent any question, and use a placeholder name for the data type of the `answer` property.

The actual data type, `String`, `Int`, `Boolean`, etc, is specified when that class is instantiated. Wherever the placeholder name is used, the data type passed into the class is used instead.



A generic data type is provided when instantiating a class, so it needs to be defined as part of the class signature. The placeholder name can then be used wherever you use a real data type within the class, such as for a property.



How would your class ultimately know which data type to use? The data type that the generic type uses is passed as a parameter in angle brackets when you instantiate the class.

After the class name comes a left-facing angle bracket (`<`), followed by the actual data type, `String`, `Boolean`, `Int`, etc., followed by a right-facing bracket (`>`).

> 💡 The data type of the value that you pass in for the generic property must match the data type in the angle brackets.

Refactor Your Code to Use Generics.

```
class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: String
)
```

Example instantiations:

```
fun main() {
    val question1 = Question<String>("Quoth the raven ___", "
    val question2 = Question<Boolean>("The sky is green. True
    val question3 = Question<Int>("How many days are there be
}
```

Use an `enum` Class.

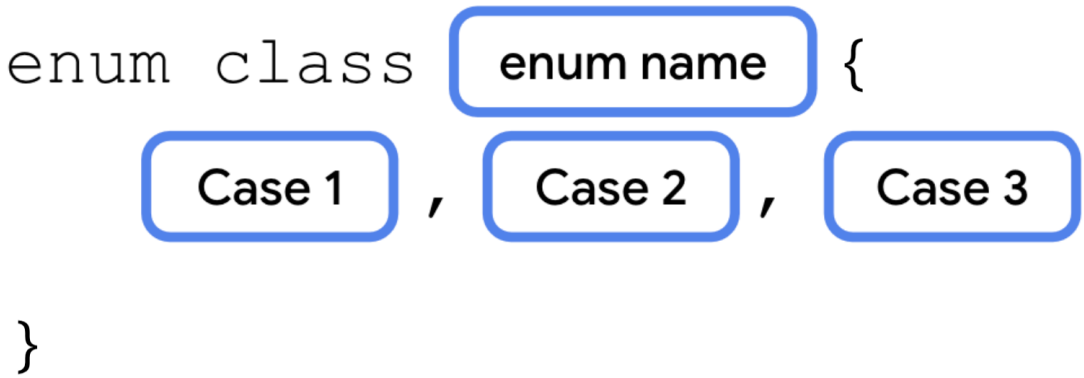
In the previous section, you defined a difficulty property with three possible values: "easy", "medium", and "hard". While this works, there are a couple of problems.

1. If you accidentally mistype one of the 3 possible strings, you could introduce bugs.
2. If the values change, for example, `"medium"` is renamed to `"average"`, then you need to update all usages of the string.
3. There's nothing stopping you or another developer from accidentally using a different string that isn't one of the three valid values.
4. The code is harder to maintain if you add more difficulty levels.

Kotlin helps you address these problems with a special type of class called an *enum class*.

> 💡 An enum class is used to create types with a limited set of possible values.

In the real world, for example, the 4 cardinal directions - north, south, east, and west - could be represented by an enum class. There's no need, and the code shouldn't allow, for the use of any additional directions. The syntax for an enum class is:



Each possible value of an enum is called an *enum constant*. Enum constants are placed inside the curly braces separated by commas.

> 💡 The convention is to capitalize every letter in the constant name.

You refer to enum constants using the dot operator.

Use an `enum` Constant.

```
enum class Difficulty {
    EASY, MEDIUM, HARD
}
```

```
class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
)
```

```
val question1 = Question<String>("Quoth the raven ___", "neve
val question2 = Question<Boolean>("The sky is green. True or
val question3 = Question<Int>("How many days are there betwee
```

Use a `data` Class.

💡 Classes like the `Question` class only contain data. They don't have any methods that perform an action. These can be defined as a data class.

Defining a class as a data class allows the Kotlin compiler to make certain assumptions, and to automatically implement some methods. When you use a data class, `toString()` and other methods are implemented automatically based on the class's properties.

Convert `Question` to a `data` Class.

```
data class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
)

fun main() {
    val question1 = Question<String>("Quoth the raven ___", "
    val question2 = Question<Boolean>("The sky is green. True
    val question3 = Question<Int>("How many days are there be
    println(question1.toString())
}

// Output is:
// Question(questionText=Quoth the raven ___, answer=nevermor
```

Use a Singleton Object.

There are many scenarios where you want a class to only have one instance. For example:

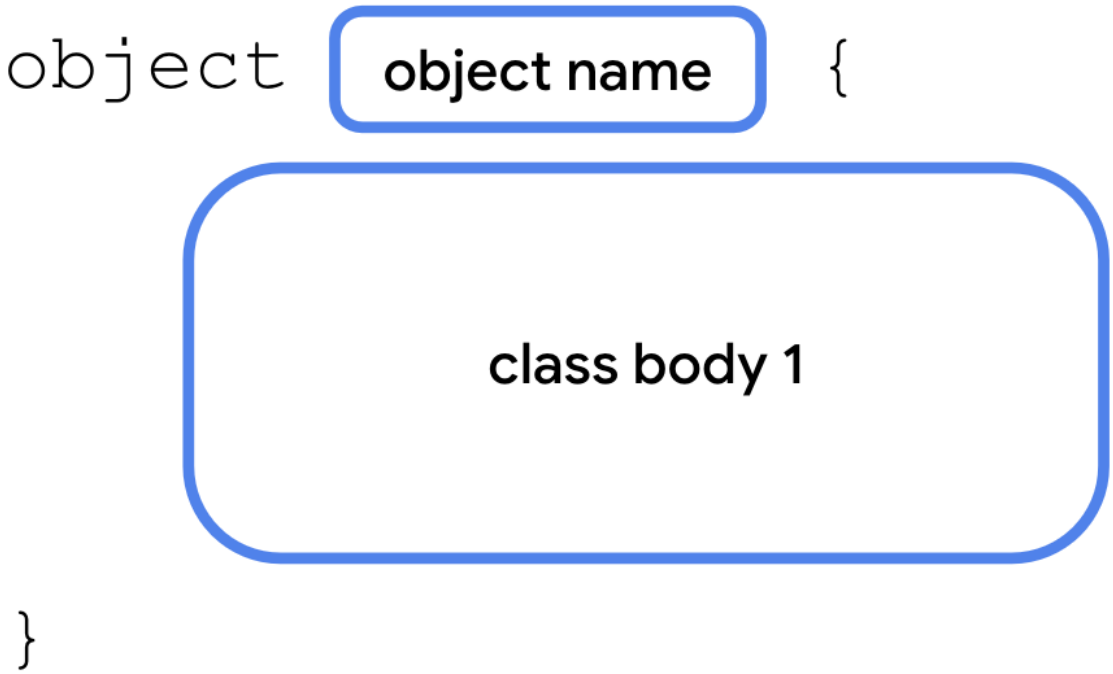
1. Player stats in a mobile game for the current user.
2. Interacting with a single hardware device, like sending audio through a speaker.
3. An object to access a remote data source (such as a Firebase database).
4. Authentication, where only one user should be logged in at a time.

You can clearly communicate in your code that an object should have only one instance by defining it as a singleton.

💡 A singleton is a class that can only have a single instance.

Kotlin provides a special construct, called an *object*, that can be used to make a singleton class.

Define a Singleton Object.



The syntax for an object is similar to that of a class. Simply use the `object` keyword instead of the `class` keyword. A singleton object can't have a constructor as you can't create instances directly. Instead, all the properties are defined within the curly braces and are given an initial value.

For a quiz, it would be great to have a way to keep track of the total number of questions, and the number of questions the student answered so far. You'll only need 1 instance of this class to exist, so instead of declaring it as a class, declare it as a singleton object.

```
object StudentProgress {
    var total: Int = 10
    var answered: Int = 3
}
```

Access a Singleton Object.

You can't create an instance of a singleton object directly. *How then are you able to access its properties?* Because there's only one instance of `StudentProgress` in existence at one time, you access its properties by referring to the name of the object itself, followed by the dot operator (`.`), followed by the property name.


```
fun main() {
    println("${StudentProgress.answered} of ${StudentProgress
}
// 3 of 10 answered.
```

Declare Objects as Companion Objects.

Classes and objects in Kotlin can be defined inside other types, and can be a great way to organize your code. You can define a singleton object inside another class using a *companion object*.

A companion object allows you to access its properties and methods from inside the class, if the object's properties and methods belong to that class, allowing for more concise syntax.

To declare a companion object, simply add the `companion` keyword before the `object` keyword. You'll create a new class called `Quiz` to store the quiz questions, and make `StudentProgress` a companion object of the `Quiz` class.

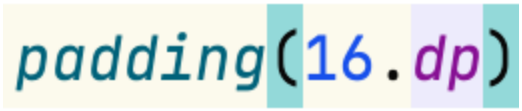
```
class Quiz {
    val question1 = Question<String>("Quoth the raven ___", "
    val question2 = Question<Boolean>("The sky is green. True
    val question3 = Question<Int>("How many days are there be

    companion object StudentProgress {
        var total: Int = 10
        var answered: Int = 3
    }
}
```

Update the call to `println()` to reference the properties with `Quiz.answered` and `Quiz.total`. Even though these properties are declared in the `StudentProgress` object, they can be accessed with dot notation using only the name of the `Quiz` class.

```
fun main() {
    println("${Quiz.answered} of ${Quiz.total} answered.")
}

// Output:
// 3 of 10 answered.
```

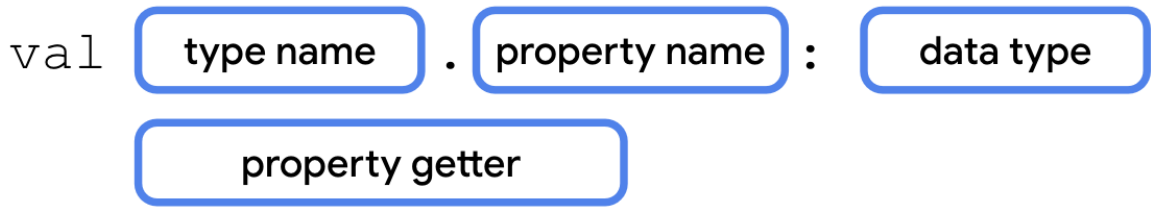
Extend Classes with New Properties and Methods.

When working with Compose, you may have noticed some interesting syntax when specifying the size of UI elements. Numeric types, such as `Double`, appear to have properties like `dp` and `sp` specifying dimensions. Why would the designers of the Kotlin language include properties and functions on built-in data types, specifically for building Android UI?



Add an Extension Property.

To define an extension property, add the type name and a dot operator before the variable name.



We can refactor the code in the `main()` function to print the quiz progress with an extension property.

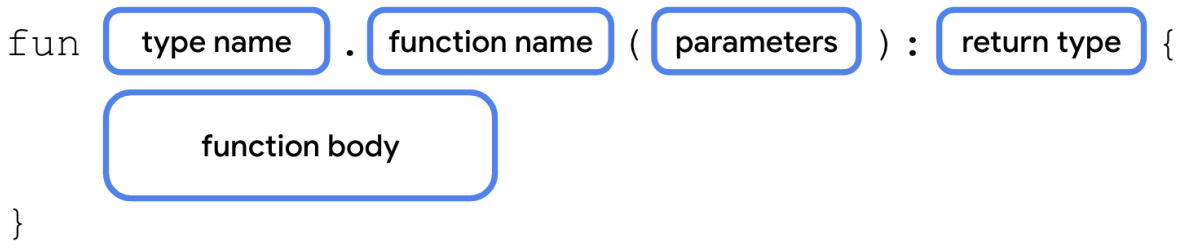
```
val Quiz.StudentProgress.progressText: String
    get() = "${answered} of ${total} answered"
```

Replace the code in the `main()` function with code that prints `progressText`. Because this is an extension property of the companion object, you can access it with dot notation using the name of the class, `Quiz`.

```
fun main() {
    println(Quiz.progressText)
}

// Output:
// 3 of 10 answered.
```

Add an Extension Function.

To define an extension function, add the type name and a dot operator (`.`) before the function name.



You'll add an extension function to output the quiz progress as a progress bar. Since you can't actually make a progress bar in the Kotlin playground, you'll print out a retro-style progress bar using text!

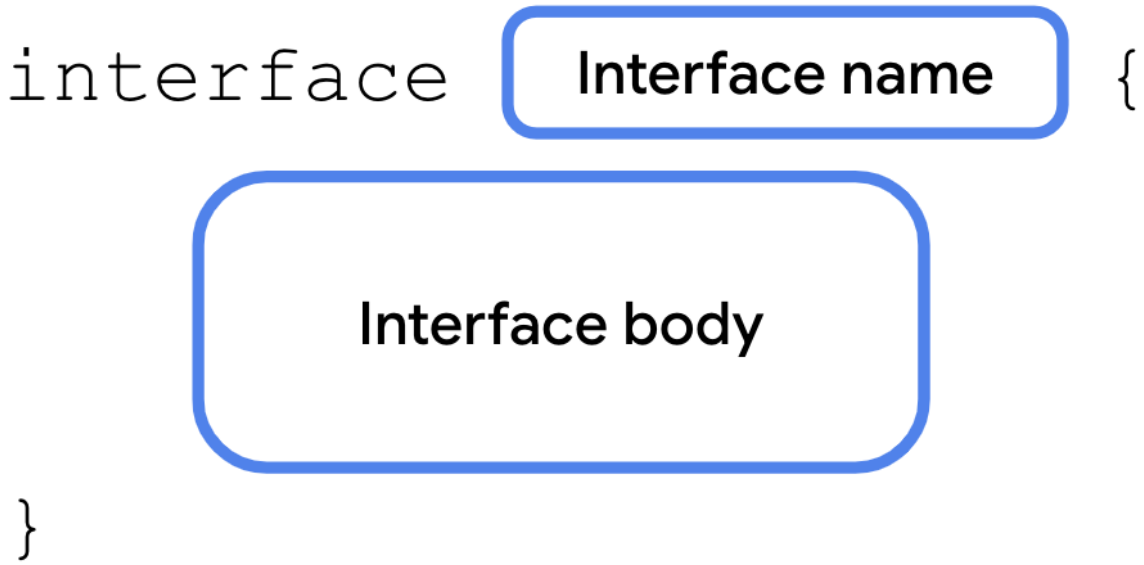
```
fun Quiz.StudentProgress.printProgressBar() {
        // Print out the ▓ character, answered number of time
    repeat(Quiz.answered) { print("▓") }
    repeat(Quiz.total - Quiz.answered) { print("▒") }
}

fun main() {
    Quiz.printProgressBar()
}

// Output:
// ▓▓▓▒▒▒▒▒▒▒
// 3 of 10 answered.
```

Rewrite Extension Functions Using Interfaces.

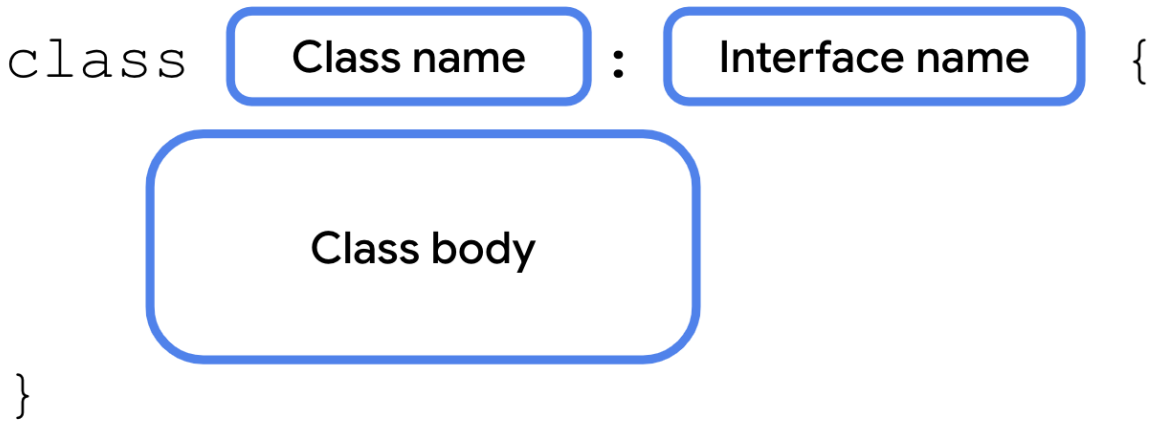
While this is a great way to add functionality to one class that's already defined, extending a class isn't always necessary if you have access to the source code. There are also situations where you don't know what the implementation should be, only that a certain method or property should exist.

💡 If you need multiple classes to have the same additional properties and methods, perhaps with differing behavior, you can define these properties and methods with an *interface*.

An interface is defined using the `interface` keyword, followed by a name in UpperCamelCase, followed by opening and closing curly braces. Within the curly braces, you can define any method signatures or get-only properties that any class conforming to the interface must implement.



An interface is a contract. A class that conforms to an interface is said to extend the interface. A class can declare that it would like to extend an interface using a colon (`:`), followed by a space, followed by the name of the interface.



In return, the class must implement all properties and methods specified in the interface. This lets you easily ensure that any class that needs to extend the interface implements the exact same methods with the exact same method signature.

```
interface ProgressPrintable {
        val progressText: String
        fun printProgressBar()
}
```

In the `Quiz` class, add a property named `progressText`, as specified in the interface. Because the property comes from `ProgressPrintable`, precede `val` with the override keyword.

```
class Quiz : ProgressPrintable {
    override val progressText: String
        get() = "${answered} of ${total} answered"

    override fun printProgressBar() {
            repeat(Quiz.answered) { print("▓") }
            repeat(Quiz.total - Quiz.answered) { print("▒") }
            println()
            println(progressText)
        }
}

fun main() {
    Quiz().printProgressBar()
}

// Output:
// ▓▓▓▒▒▒▒▒▒▒
// 3 of 10 answered.
```

Use Scope Functions to Access Class Properties and Methods.

> 💡 Scope functions allow you to concisely access properties and methods from a class without having to repeatedly access the variable name.

These are called scope functions because the body of the function passed in takes on the scope of the object that the scope function is called with. For example, some scope functions allow you to access the properties and methods in a class, as if the functions were defined as a method of that class. This can make your code more readable by allowing you to omit the object name when including it is redundant.

Replace Long Object Names Using `let()`.

> 💡 The `let()` function allows you to refer to an object in a lambda expression using the identifier it, instead of the object's actual name.

This can help you avoid using a long, more descriptive object name repeatedly when accessing more than one property. The `let()` function is an extension function that can be called on any Kotlin object using dot notation.

Add the following code that prints the question's `questionText`, `answer`, and `difficulty`. While multiple properties are accessed for `question1`, `question2`, and `question3`, the entire variable name is used each time. If the variable's name changed, you'd need to update every usage.

```
fun printQuiz() {
    println(question1.questionText)
    println(question1.answer)
    println(question1.difficulty)
    println()
    println(question2.questionText)
    println(question2.answer)
    println(question2.difficulty)
    println()
```

```
    println(question3.questionText)
    println(question3.answer)
    println(question3.difficulty)
    println()
}
```

Wrap the code that accesses the `questionText`, `answer`, and `difficulty` properties with a call to the `let()` function on `question1`, `question2`, and `question3`. Replace the variable name in each lambda expression with `it`.

```
fun printQuiz() {
    question1.let {
        println(it.questionText)
        println(it.answer)
        println(it.difficulty)
    }
    println()
    question2.let {
        println(it.questionText)
        println(it.answer)
        println(it.difficulty)
    }
    println()
    question3.let {
        println(it.questionText)
        println(it.answer)
        println(it.difficulty)
    }
    println()
}
```

Call an Object's Methods Without a Variable Using `apply()`.

One of the great features of scope functions is that you can invoke them on an object before assigning that object to a variable. For instance, the `apply()`

function is an extension function that can be called on an object using dot notation.

> 💡 The `apply()` function returns a reference to the object, allowing it to be stored in a variable.

Invoke `apply()` after the closing parenthesis when instantiating the `Quiz` class. You can omit the parentheses for `apply()` and use trailing lambda syntax.

```
val quiz = Quiz().apply {
}
```

Place the call to `printQuiz()` inside the lambda expression. You no longer need to reference the `quiz` variable or use dot notation.

```
val quiz = Quiz().apply {
    printQuiz()
}
```

The `apply()` function returns the instance of the `Quiz` class, but since you're not using it anywhere, you can remove the `quiz` variable. With the `apply()` function, you can call methods on the `Quiz` instance without needing a variable.

```
Quiz().apply {
    printQuiz()
}
```

▼ Use Collections in Kotlin.

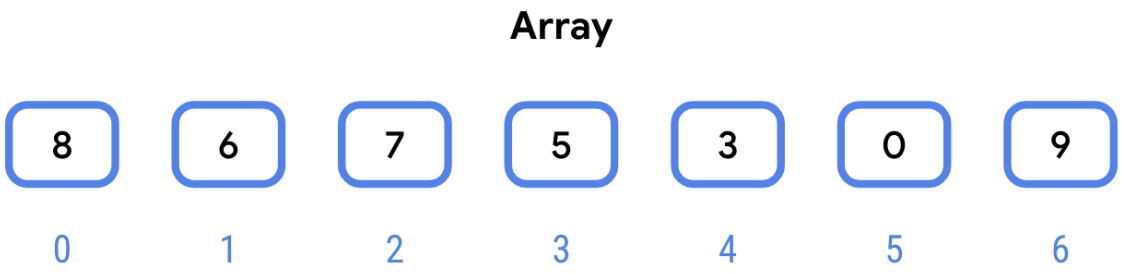
To build apps involving arbitrary amounts of data, collections are essential. Collection types (sometimes called data structures) let you store multiple values, typically of the same data type, in an organized way.

A collection might be an ordered list, a grouping of unique values, or a mapping of values of one data type to values of another. The ability to effectively use

collections enables you to implement common features of Android apps, such as scrolling lists, as well as solve a variety of real-life programming problems that involve arbitrary amounts of data.

Arrays in Kotlin.

An array is a sequence of values that all have the same data type. An array contains multiple values called *elements*, or sometimes, *items*. The elements in an array are ordered and are accessed with an index.



💡 Accessing an array element by its index is fast. You can access any random element of an array by its index and expect it to take about the same amount of time to access any other random element. This is why it's said that arrays have *random access*.

💡 An array has a fixed size. This means that you can't add elements to an array beyond this size. Trying to access the element at index 100 in a 100 element array will throw an exception because the highest index is 99 (remember that the first index is 0, not 1). You can, however, modify the values at indexes in the array.

To declare an array in code, you use the `arrayOf()` function.

```
val variable name = arrayOf<data type>(element1, element2, ...)
```

```
val rockPlanets = arrayOf<String>("Mercury", "Venus", "Earth"
val gasPlanets = arrayOf<String>("Jupiter", "Saturn", "Uranus
```

You can add 2 arrays together.

```
val solarSystem = rockPlanets + gasPlanets
```

Access an Element in an Array.

This is called subscript syntax. It consists of three parts:

- The name of the array.
- An opening (`[`) and closing (`]`) square bracket.
- The index of the array element in the square brackets.

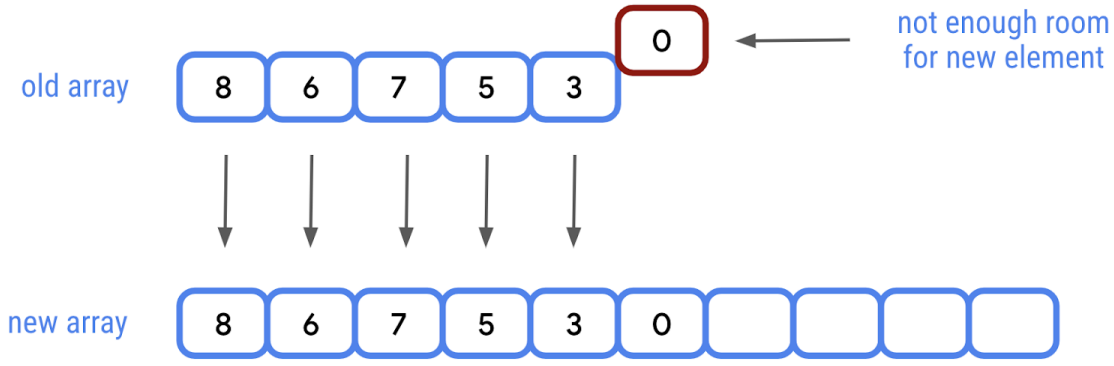
You can also set the value of an array element by its index.

If you want to make an array larger than it already is, you need to create a new array. Define a new variable called `newSolarSystem` as shown. This array can store nine elements, instead of eight.

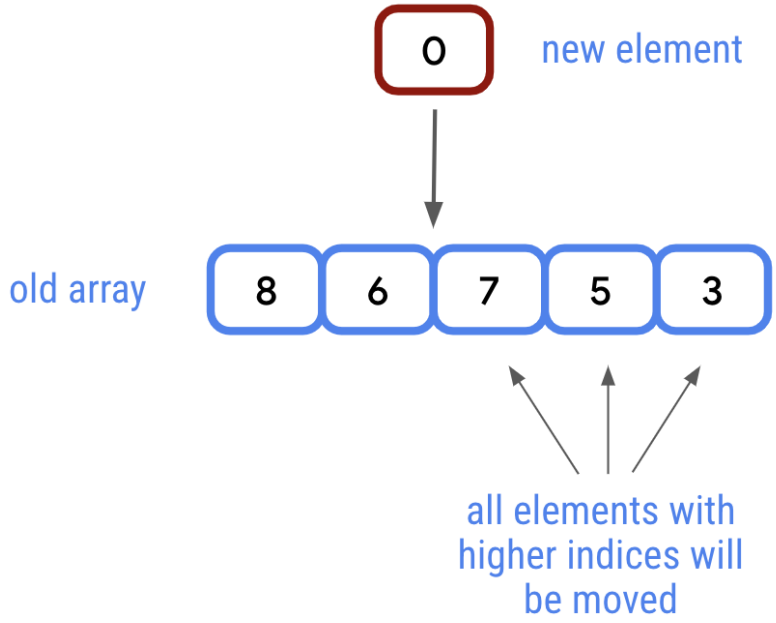
Lists.

A list is an ordered, resizable collection, typically implemented as a resizable array. When the array is filled to capacity and you try to insert a new element,

the array is copied to a new bigger array.



With a list, you can also insert new elements between other elements at a specific index.



`List` and `MutableList`.

The collection types you'll encounter in Kotlin implement one or more interfaces.

So what do `List` and `MutableList` do?

- `List` is an interface that defines properties and methods related to a read-only ordered collection of items.
- `MutableList` extends the `List` interface by defining methods to modify a list, such as adding and removing elements.

Like `arrayOf()`, the `listOf()` function takes the items as parameters, but returns a `List` rather than an array.

```
val solarSystem = listOf("Mercury", "Venus", "Earth", "Mars",
println(solarSystem.size) // 8
```

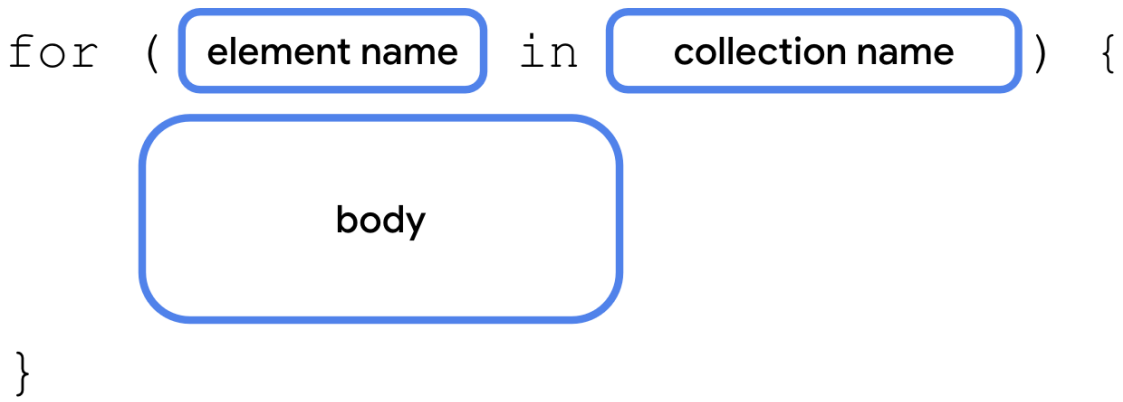
Access Elements From a List.

Like an array, you can access an element at a specific index from a `List` using subscript syntax. You can also use the `get()` method. Subscript syntax and the `get()` method take an Int as a parameter and return the element at that index.

The `indexOf()` method searches the list for a given element (passed in as an argument), and returns the index of the first occurrence of that element. If the element doesn't occur in the list, it returns `-1`.

```
println(solarSystem.indexOf("Earth"))
```

Iterate Over List Elements Using a `for` loop.



```
for (planet in solarSystem) {
    println(planet)
}
```

Add Elements to a List.

The ability to add, remove, and update elements in a collection is exclusive to classes that implement the `MutableList` interface. If you were keeping track of newly discovered planets, you'd likely want the ability to frequently add elements to a list. You need to specifically call the `mutableListOf()` function, instead of `listOf()`, when creating a list you wish to add and remove elements from.

There are two versions of the `add()` function:

- The first `add()` function has a single parameter of the type of element in the list and adds it to the end of the list.
- The other version of `add()` has two parameters. The first parameter corresponds to an index at which the new element should be inserted. The second parameter is the element being added to the list.

```
val solarSystem = mutableListOf("Mercury", "Venus", "Earth",
solarSystem.add("Pluto")
solarSystem.add(3, "Theia")
```

Update Elements at a Specific Index.

```
solarSystem[3] = "Future Moon"
```

Remove Elements from a List.

Elements are removed using the `remove()` or `removeAt()` method. You can either remove an element by passing it into the `remove()` method or by its index using `removeAt()`.

```
// This should remove Pluto from the list.
solarSystem.removeAt(9)
```

Call `remove()` on `solarSystem`, passing in `"Future Moon"` as the element to remove. This should search the list, and if a matching element is found, it will be removed.

```
solarSystem.remove("Future Moon")
```

`List` provides the `contains()` method that returns a `Boolean` if an element exists in a list. Print the result of calling `contains()` for `"Pluto"`.

```
println(solarSystem.contains("Pluto"))
```

An even more concise syntax is to use the `in` operator. You can check if an element is in a list using the element, the `in` operator, and the collection. Use the `in` operator to check if the `solarSystem` contains `“Future Moon”`.

```
println("Future Moon" in solarSystem)
```

Sets.

A set is a collection that does not have a specific order and does not allow duplicate values.

How is a collection like this possible? The secret is a hash code. A hash code is an integer produced by the `hashCode()` method of any Kotlin class. A small change to the object, such as adding one character to a string, results in a vastly different hash value.

```
"Kotlin".hashCode()     // -204170231
"Kotlin!".hashCode()    // 1131585312
```

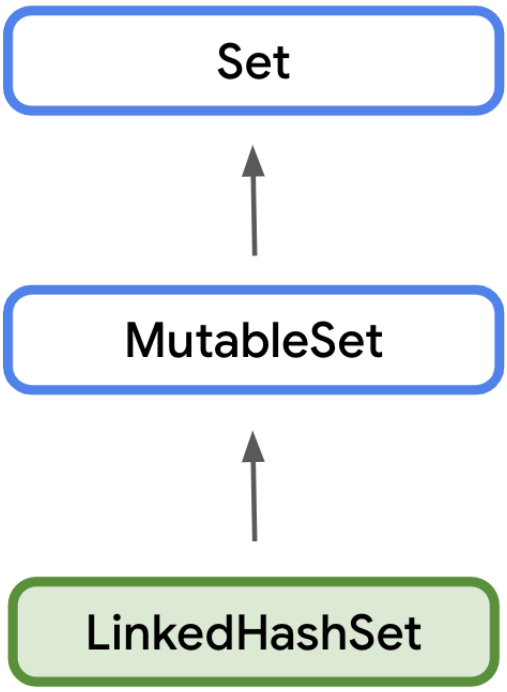
Sets have 2 important properties:

> 💡 Searching for a specific element in a set is fast—compared with lists—especially for large collections. While the `indexOf()` of a `List` requires checking each element from the beginning until a match is found, on average, it takes the same amount of time to check if an element is in a set, whether it's the first element or the hundred thousandth.

> 💡 Sets tend to use more memory than lists for the same amount of data, since more array indices are often needed than the data in the set.

The benefit of sets is ensuring uniqueness. If you were writing a program to keep track of newly discovered planets, a set provides a simple way to check if a planet has already been discovered. With large amounts of data, this is often preferable to checking if an element exists in a list, which requires iterating over all the elements.

Like `List` and `MutableList`, there's both a `Set` and a `MutableSet`. `MutableSet` implements `Set`, so any class implement `MutableSet` needs to implement both.

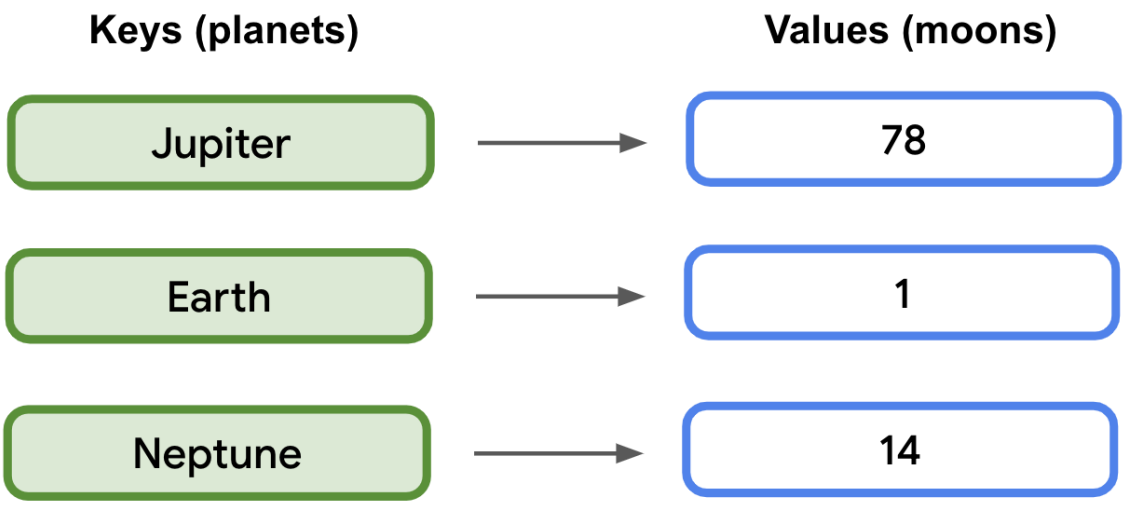


Use a `MutableSet` in Kotlin.

```
val solarSystem = mutableSetOf("Mercury", "Venus", "Earth", "
println(solarSystem.size)
solarSystem.add("Pluto")
println(solarSystem.size)
println(solarSystem.contains("Pluto"))
println("Pluto" in solarSystem)
solarSystem.remove("Pluto")
```

Map Collection.

A Map is a collection consisting of keys and values. It's called a map because unique keys are *mapped* to other values. A key and its accompanying value are often called a `key-value pair`.



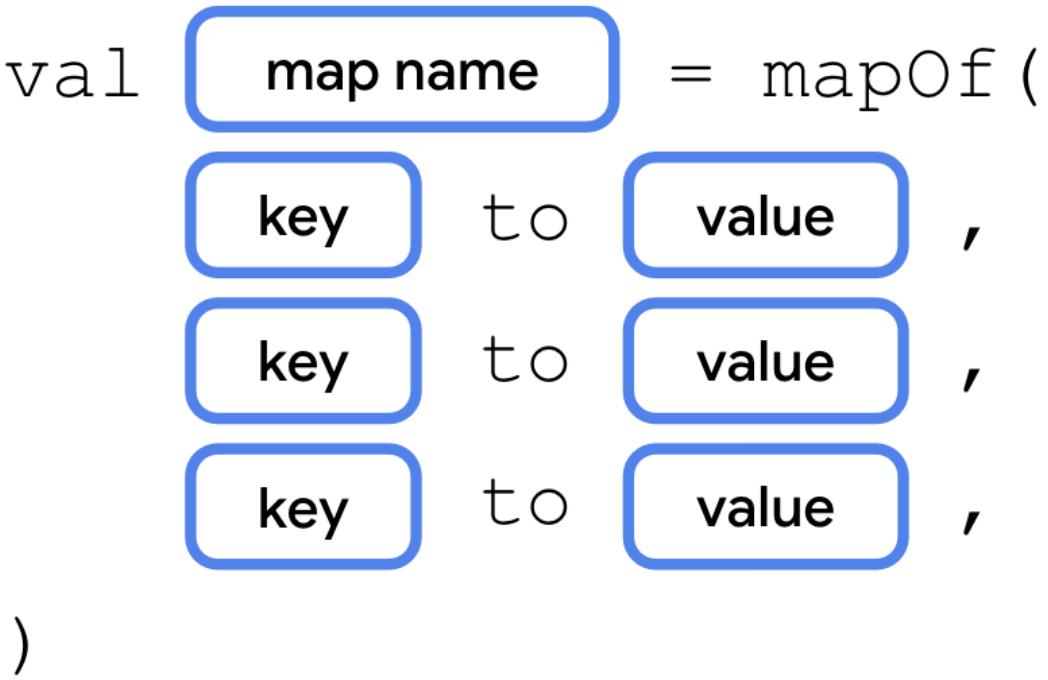
A map's keys are unique. A map's values, however, are not.

Accessing a value from a map by its key is generally faster than searching through a large list, such as with `indexOf()`.

Maps can be declared using the `mapOf()` or `mutableMapOf()` function. Maps require 2 generic types separated by a comma - one for the keys and another for the values.

```
mutableMapOf<key type, value type>()
```

A map can also use type inference if it has initial values. To populate a map with initial values, each key value pair consists of the key, followed by the `to` operator, followed by the value. Each pair is separated by a comma.




```
val solarSystem = mutableMapOf(
    "Mercury" to 0,
    "Venus" to 0,
    "Earth" to 1,
    "Mars" to 2,
    "Jupiter" to 79,
    "Saturn" to 82,
    "Uranus" to 27,
    "Neptune" to 14
)
println(solarSystem.size)
solarSystem["Pluto"] = 5 // mapName[key] = value
println(solarSystem["Pluto"]) // 5
```

You can also access values with the `get()` method. Whether you use subscript syntax or call `get()`, it's possible that the key you pass in isn't in the map. If there isn't a key-value pair, it will return null. Print the number of moons for `"Theia"`.

```
println(solarSystem.get("Theia")) // null
```

The `remove()` method removes the key-value pair with the specified key. It also returns the removed value, or `null`, if the specified key isn't in the map.

```
solarSystem.remove("Pluto")
```

▼ Higher-Order Functions with Collections.

`forEach()` and String Templates with Lambdas.

In the following examples, you'll take a `List` representing a bakery's cookie menu and use higher-order functions to format the menu in different ways.

```
class Cookie(
    val name: String,
    val softBaked: Boolean,
    val hasFilling: Boolean,
    val price: Double
)

val cookies = listOf(
    Cookie(
        name = "Chocolate Chip",
        softBaked = false,
        hasFilling = false,
        price = 1.69
    ),
    Cookie(
        name = "Banana Walnut",
        softBaked = true,
        hasFilling = false,
        price = 1.49
    ),
    Cookie(
        name = "Vanilla Creme",
```

```
        softBaked = false,
        hasFilling = true,
        price = 1.59
    ),
    Cookie(
        name = "Chocolate Peanut Butter",
        softBaked = false,
        hasFilling = true,
        price = 1.49
    ),
    Cookie(
        name = "Snickerdoodle",
        softBaked = true,
        hasFilling = false,
        price = 1.39
    ),
    Cookie(
        name = "Blueberry Tart",
        softBaked = true,
        hasFilling = true,
        price = 1.79
    ),
    Cookie(
        name = "Sugar and Sprinkles",
        softBaked = false,
        hasFilling = false,
        price = 1.39
    )
)

fun main() {

}
```

Loop Over a List with `forEach()`.

The `forEach()` function executes the function passed as a parameter once for each item in the collection. This works like a `for` loop. The lambda is executed for the first element, then the second element, and so on, until it's executed for each element in the collection.

```
forEach(action: (T) -> Unit)
```

`T` corresponds to whatever data type the collection contains. Because the lambda takes a single parameter, you can omit the name and refer to the parameter with `it`.

```
fun main() {
    cookies.forEach {
        println("Menu item: $it")
    }
}
```

All that prints is the name of the type (`Cookie`), and a unique identifier for the object, but not the contents of the object.

```
Menu item: Cookie@5a10411
Menu item: Cookie@68de145
Menu item: Cookie@27fa135a
Menu item: Cookie@46f7f36a
Menu item: Cookie@421faab1
Menu item: Cookie@2b71fc7e
Menu item: Cookie@5ce65a89
```

Embed Expressions in Strings.

When you were first introduced to string templates, you saw how the dollar symbol (`$`) could be used with a variable name to insert it into a string.

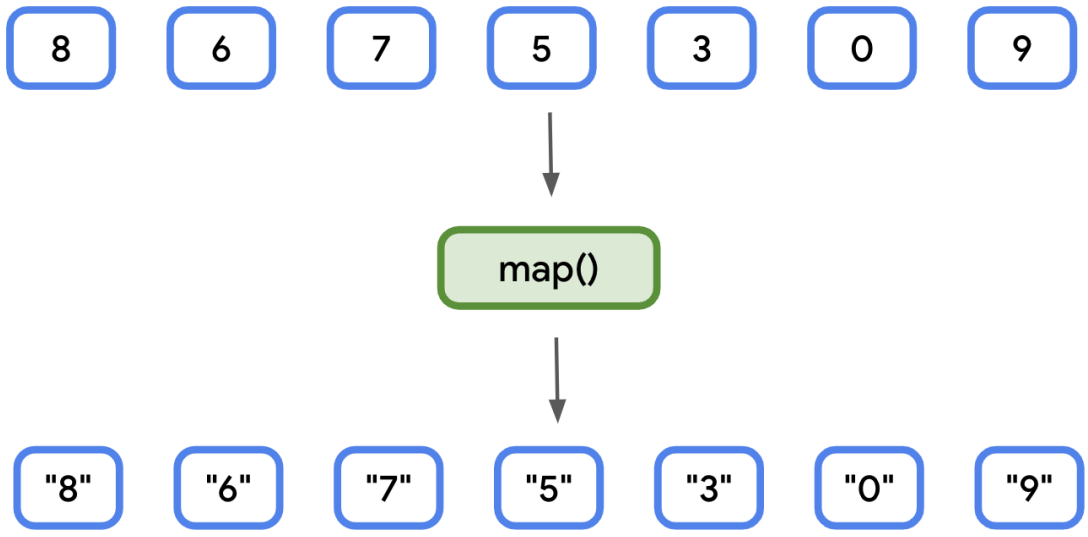
However, this doesn't work as expected when combined with the dot operator (`.`) to access properties.

To access properties and embed them in a string, you need an expression. You can make an expression part of a string template by surrounding it with curly braces. The lambda expression is placed between the opening and closing curly braces. You can access properties, perform math operations, call functions, etc. and return the value of the lambda is inserted into the string.

```
cookies.forEach {
    println("Menu item: ${it.name}")
}
```

`map()`.

The `map()` function lets you transform a collection into a new collection with the same number of elements. For example, it could transform a `List<Cookie>` into a `List<String>` only containing the cookie's name, provided you tell the `map()` function how to create a `String` from each `Cookie` item.

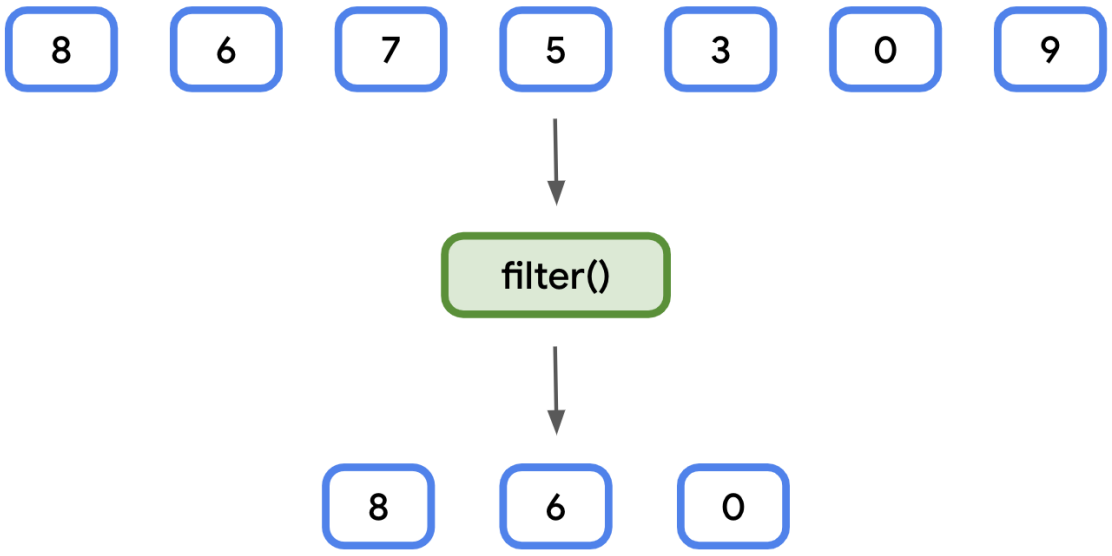



```
val fullMenu = cookies.map {
    "${it.name} - $${it.price}"
```

```
}
println("Full menu:")
fullMenu.forEach {
    println(it)
}

// Output:
// Full menu:
// Chocolate Chip - $1.69
// Banana Walnut - $1.49
// Vanilla Creme - $1.59
// Chocolate Peanut Butter - $1.49
// Snickerdoodle - $1.39
// Blueberry Tart - $1.79
// Sugar and Sprinkles - $1.39
```

`filter()`.

The `filter()` function lets you create a subset of a collection. For example, if you had a list of numbers, you could use `filter()` to create a new list that only contains numbers divisible by 2.



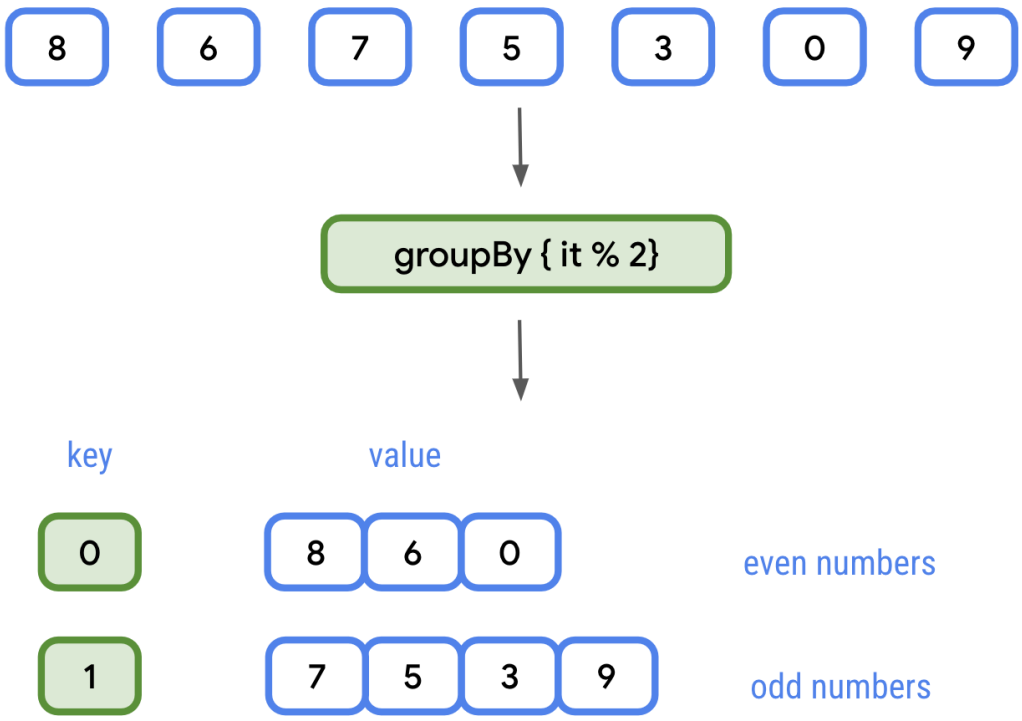
Unlike `map()`, the resulting collection also has the same data type, so filtering a `List<Cookie>` will result in another `List<Cookie>`.

```
val softBakedMenu = cookies.filter {
    it.softBaked
}
println("Soft cookies:")
softBakedMenu.forEach {
    println("${it.name} - $${it.price}")
}
// Output:
// Soft cookies:
// Banana Walnut - $1.49
// Snickerdoodle - $1.39
// Blueberry Tart - $1.79
```

`groupBy()`.

The `groupBy()` function can be used to turn a list into a map, based on a function. Each unique return value of the function becomes a key in the resulting map. The values for each key are all the items in the collection that produced that unique return value.

You can check if a number is odd or even by dividing it by `2` and checking if the remainder is `0` or `1`. If the remainder is `0`, the number is even. Otherwise, if the remainder is `1`, the number is odd. This can be achieved with the modulo operator (`%`). The modulo operator divides the dividend on the left side of an expression by the divisor on the right.



Instead of returning the result of the division, like the division operator (`/`), the modulo operator returns the remainder. This makes it useful for checking if a number is even or odd. The `groupBy()` function is called with the following lambda expression: `{ it % 2 }`.

The resulting map has two keys: `0` and `1`. Each key has a value of type `List<Int>`. The list for key `0` contains all even numbers, and the list for key `1` contains all odd numbers.

> 💡 A real-world use case might be a photos app that groups photos by the subject or location where they were taken. For our bakery menu, let's group the menu by whether or not a cookie is soft baked.

Pass in a lambda expression that returns `it.softBaked`. The return type will be `Map<Boolean, List<Cookie>>`.

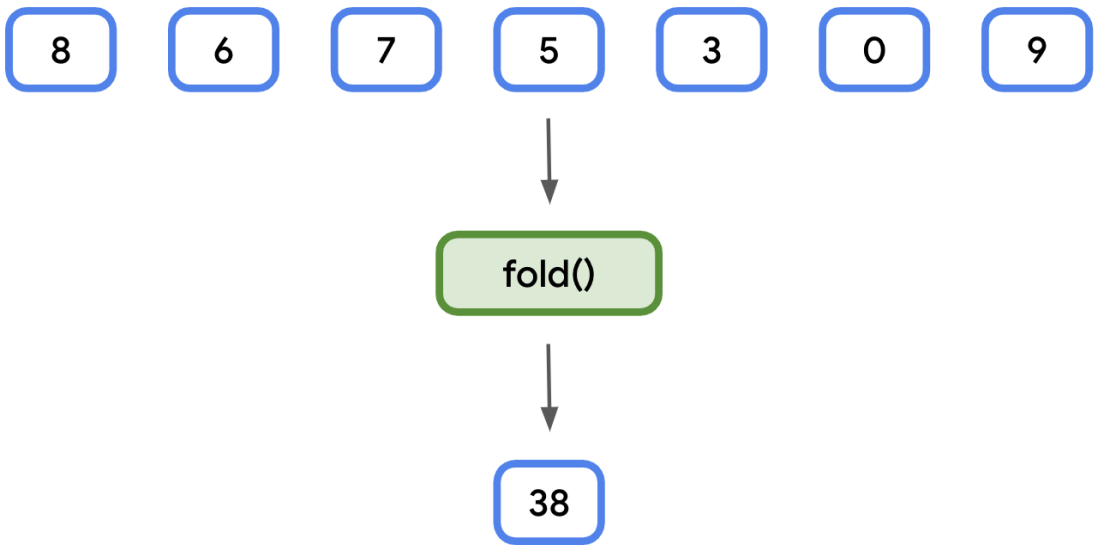
```
val groupedMenu = cookies.groupBy { it.softBaked }
```

Create a `softBakedMenu` variable containing the value of `groupedMenu[true]`, and a `crunchyMenu` variable containing the value of `groupedMenu[false]`. Because the result of subscripting a `Map` is nullable, you can use the Elvis operator (`?:`) to return an empty list.

```
val softBakedMenu = groupedMenu[true] ?: listOf()
val crunchyMenu = groupedMenu[false] ?: listOf()
```

`fold()`.

The `fold()` function is used to generate a single value from a collection. This is most commonly used for things like calculating a total of prices, or summing all the elements in a list to find an average.



The `fold()` function takes two parameters:

- An initial value. The data type is inferred when calling the function.
- A lambda expression that returns a value with the same type as the initial value.

The lambda expression additionally has two parameters:

- The first is known as the accumulator. It has the same data type as the initial value. Think of this as a running total. Each time the lambda expression is called, the accumulator is equal to the return value from the previous time the lambda was called.
- The second is the same type as each element in the collection.

```
val totalPrice = cookies.fold(0.0) {total, cookie ->
    total + cookie.price
}
```

💡 `fold()` is sometimes called `reduce()`. The `fold()` function in Kotlin works the same as the `reduce()` function found in JavaScript, Swift, Python, etc. Note that Kotlin also has its own function called `reduce()`, where the accumulator starts with the first element in the collection, rather than an initial value passed as an argument.

`sortedBy()`.

The `sort()` function could be used to sort elements in a collection but it won't work on a collection of `Cookie` objects because the `Cookie` class has several properties and Kotlin won't know which properties you want to sort by.

`sortedBy()` lets you specify a lambda that returns the property you'd like to sort by. For example, if you'd like to sort by `price`, the lambda would return `it.price`. So long as the data type of the value has a natural sort order—strings are sorted alphabetically, numeric values are sorted in ascending order—it will be sorted just like a collection of that type.

```
val alphabeticalMenu = cookies.sortedBy {
    it.name
}
```