Dart

Functional Programming

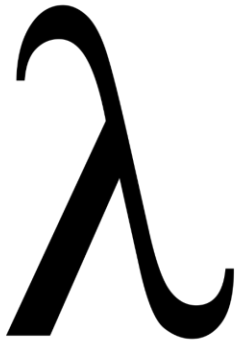
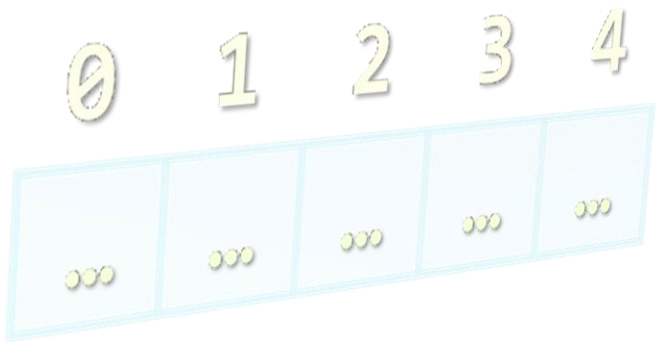


Table of Contents

Collections



Back

List

- Dart has a List**<T>** type to declare list

```
List<String> colors = ["Red", "Green", "Blue"];
var names = ["Ali", "Ahmed", "Sara"];

const nums = [2, 3, 4];
var nullNums = List<int?>.filled(10, null);

colors.forEach((color) => print(color));
names.forEach((name) => print(name));
nums.forEach((num) => print(num));
nullNums.forEach((num) => print(num));
```

List Methods

```
const nums = [2, 3, 4];

nums.add(8);
nums.insert(0, 1);
nums.removeAt(2);
nums.remove(4);
nums.removeLast();
nums.removeRange(0, 2);
nums.removeWhere((num) => num > 3);
nums.removeRange(0, nums.length);
nums.addAll([1, 2, 3]);
nums.addAll([4, 5, 6]);
```

List destructuring

- List destructuring allows you to unpack or extract values from a list and assign them to variables in a clean and concise way

```
var fruits = ["Apple", "Banana", "Cherry", "Mango", "Orange"];

// Destructuring the list
// ... is used to unpack the remaining elements
var [firstFruit, secondFruit, thirdFruit, ...others] = fruits;

print("First fruit: $firstFruit");   // Output: First fruit: Apple
print("Second fruit: $secondFruit"); // Output: Second fruit: Banana
print("Third fruit: $thirdFruit");   // Output: Third fruit: Cherry
print("Others: $others");            // Output: Others: [Mango, Orange]
```

Spread operator (...)

- Spread operator (...) allows you to include all elements of one list inside another list
  - It "spreads" the elements of a list into a new list
  - The null-aware spread operator (...?) is used when the list you're spreading might be null

```
List<String> fruits = ["Apple", "Banana"];
List<String> vegetables = ["Carrot", "Broccoli"];

List<String> food = fruits + vegetables;
print(food); // Output: [Apple, Banana, Carrot, Broccoli]

food = [...fruits, ...vegetables];
print(food); // Output: [Apple, Banana, Carrot, Broccoli]
```

Set

- *Set is same as List but does not allow duplicates*

```
final Set<String> colors = {"red", "blue", "yellow"};
colors.add("pink");  // Adding a new element
 // Won't be added again because sets don't allow duplicates
colors.add("blue");
print(colors); // Output: {red, blue, yellow, pink}
```

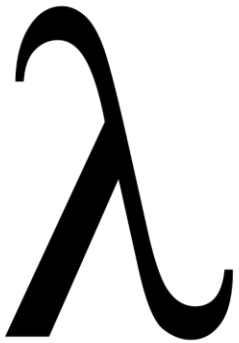
Map

- Stores keys and associated values

```
Map<int, String> languages = {
  1: "Python",
  2: "Kotlin",
  3: "Java",
};

languages.forEach((key, value) {
  print("$key => $value");
});
```

Lambda



Imperative vs. Declarative

Imperative Programming

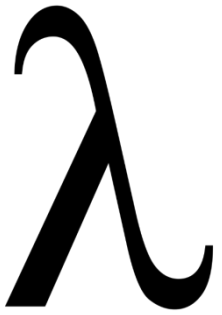
- You tell the computer **how** to perform a task

Declarative Programming

- You tell the computer **what you want**, and you let the compiler (or runtime) figure out the best way to do it. This makes the code simpler and more concise
- Also known as **Functional Programming**
- `Declarative programming using Lambdas helps us to achieve KISS`

KEEP IT SHORT & SIMPLE

What is a Lambda?



- Lambda is an **anonymous function** that you can store in a variable, pass them as parameter, or return from other function. It has:
  - Parameters
  - A body
- It **don't have a name** (anonymous method)
- It **can be passed as a parameter** to other function:
  - As *code* to be executed by the receiving function
- Concise syntax:

```
(Parameters) => Body
```

Passing Lambda as a Parameter

- Lambda expression can be passed as a parameter to methods such as ***forEach***, ***filter*** and ***map*** methods :

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
numbers.forEach((e) => print(e));
```

forEach - **Calls a Lambda on Each Element of the list**

- Left side of `=>` operator is a parameter variable
- Right side is the code to operate on the parameter and compute a result
- When using a lambda with a `List` the compiler can determine the parameter type

Lambda usage

- Allows working with collections in a **functional style**

```
bool isEven(int n) => n % 2 == 0;
void main() {
  // Range (1 to 10 inclusive)
  List<int> nums = List.generate(10, (i) => i + 1);
  // Version 1
  bool hasEvenNumber = nums.any((n) => n.isEven);
  // Verion 2
  hasEvenNumber = nums.any(isEven);

  // Version 3 - most compact
  hasEvenNumber = nums.any((n) => n % 2 == 0);
  print("Has even number: $hasEvenNumber");

  // Version 1
  List<int> evens = nums.where(isEven).toList();

  // Version 2
  evens = nums.where((n) => n % 2 == 0).toList();

  print("Even numbers: $evens");
}
```

Lambda usage

e.g. What's the average age of employees working in Doha?

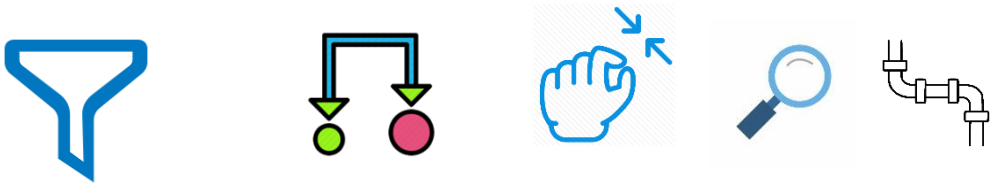
```
List<Employee> employees = [
  Employee(name: "Sara Faleh", city: "Doha", age: 30),
  Employee(name: "Mariam Saleh", city: "Istanbul", age: 22),
  Employee(name: "Ali Al-Ali", city: "Doha", age: 24),
];

// Filtering employees in "Doha", mapping their ages,
// and calculating the average
double avgAge = employees
    .where((employee) => employee.city == "Doha")
    .map((employee) => employee.age)
    .reduce((a, b) => a + b) /
    employees.where((employee) => employee.city == "Doha").length;

print("Average age of employees in Doha: $avgAge");
```

Common operations on collections

Filter, Map, Reduce, and others



Common operations on collections

`.map` 

- Applies a function to each list element

`.filter(condition)` 

- Returns a new list with the elements that satisfy the condition

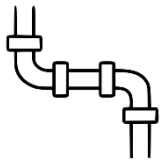
`.find(condition)` 

- Returns the first list element that satisfy the condition

`.reduce` 

- Applies an accumulator function to each element of the list to reduce them to a single value

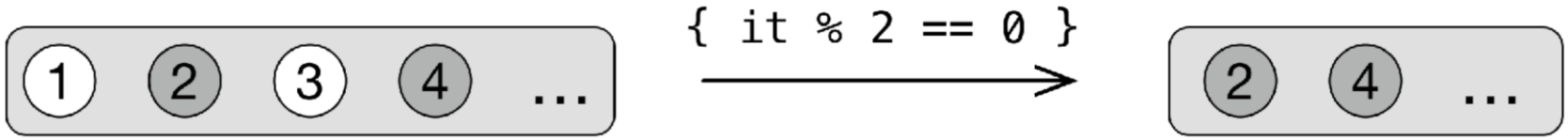
Operations Pipeline



- **A pipeline of operations:** a sequence of operations where the output of each operation becomes the input into the next
  - e.g., .filter -> .map -> .toList
- Operations are either **Intermediate** or **Terminal**
- **Intermediate operations** produce a new list as output (e.g., map, filter, …)
- **Terminal operations** are the final operation in the pipeline (e.g., find, reduce, toList …)
  - Once a terminal operation is invoked then no further operations can be performed

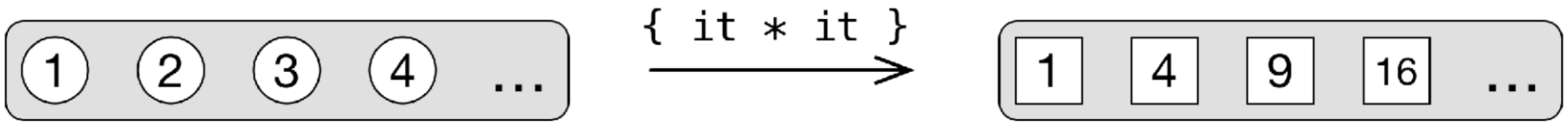
Filter

Keep elements that satisfy a condition



Map

Transform elements by applying a Lambda to each element



Reduce

Apply an accumulator function to each element of the list to reduce them to a single value

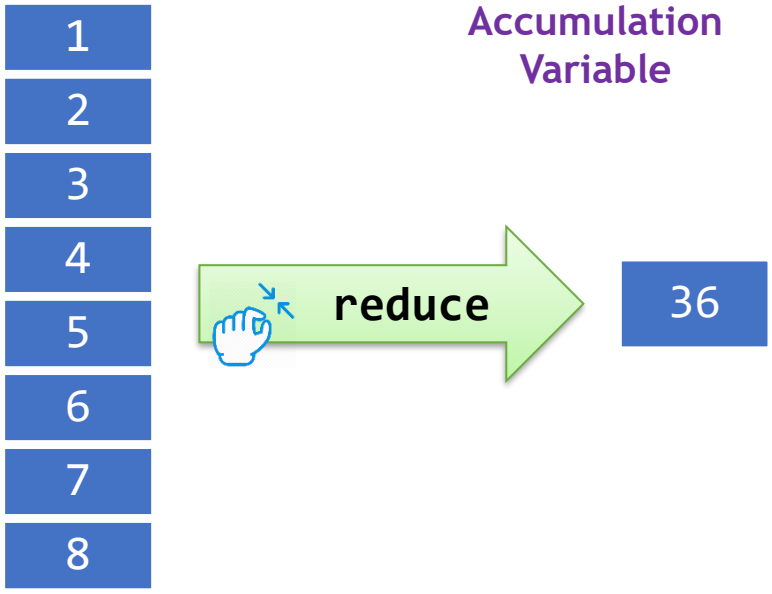
```
// Imperative
var sum = 0
for(n in numbers)
    sum += n
```

```
//Declarative
var total = numbers.reduce { sum, n -> sum + n }
//Another way with the ability to set the initial
value of sum
total = numbers.fold(0) { sum, n -> sum + n }
//Short form
total = numbers.sum()
```

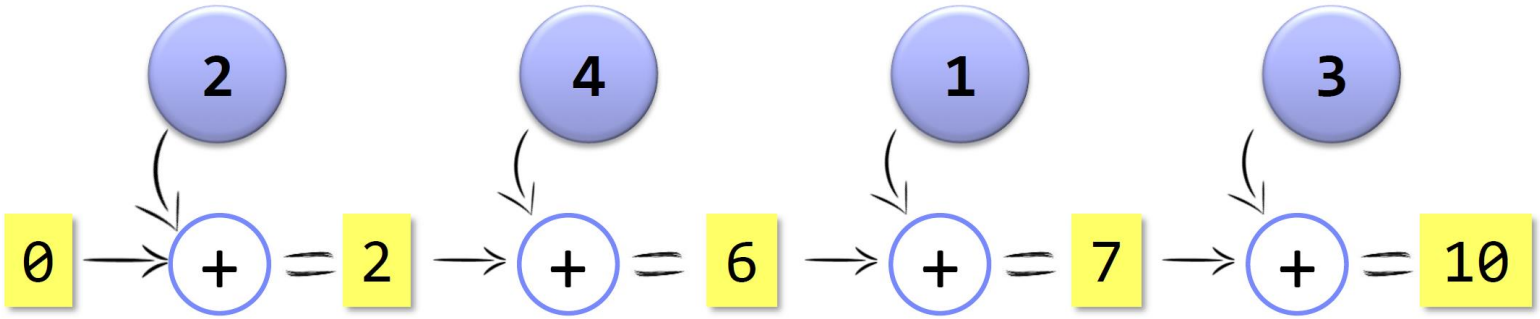
Accumulation Variable

Accumulation Lambda

Collapse the multiple elements of a list into a single element



Reduce




```
.reduce { acc, n -> acc + n }
```

Reduce is **terminal** operation that yields a single value

Convenience Reducers

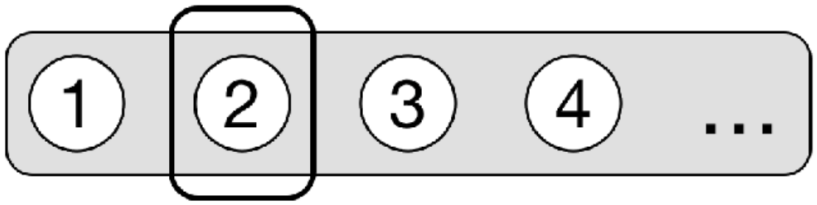
sum, average, count, min, max

- They are **terminal** operations that yield a single value

```
val nums = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val sum = nums.sum()
val count = nums.count()
val average = nums.average()
val max = nums.maxOrNull()
val min = nums.minOrNull()
```

any (all, none)

- **any** returns true if it finds an element that satisfies the lambda condition
- **all** returns false if it finds an element that fails the lambda condition
- **none** returns false if it finds an element that satisfies the lambda condition

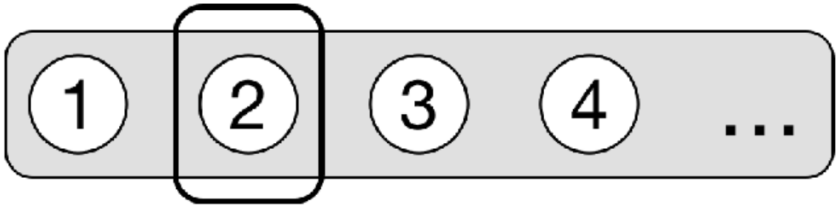



```
{ it % 2 == 0 }
```

true

find / firstWhere

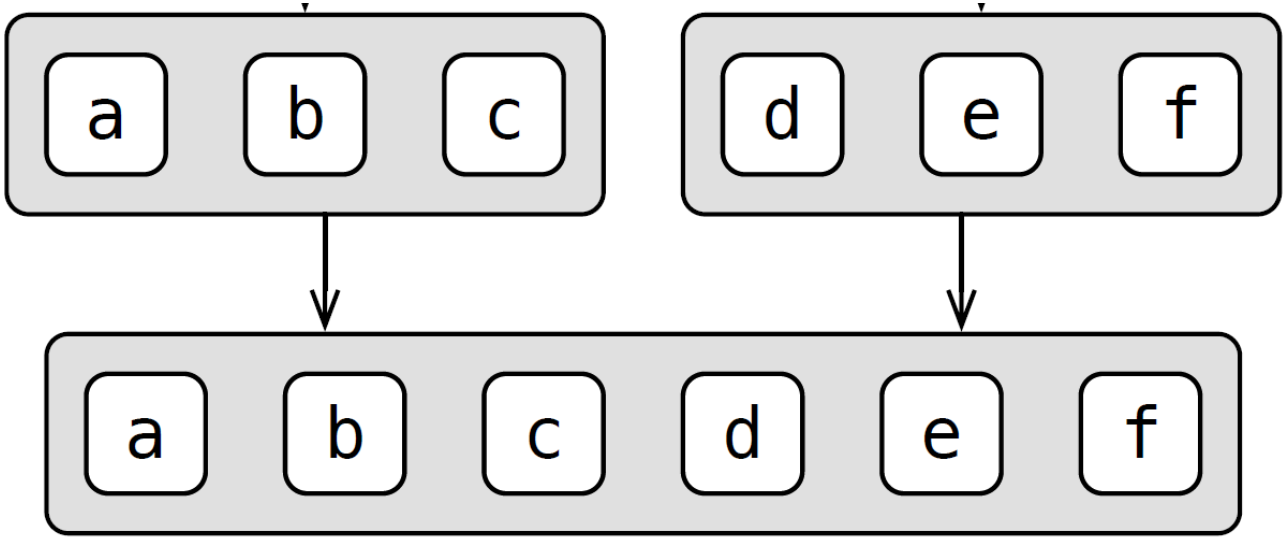
Return first element satisfying a condition




```
{ it % 2 == 0 }
```

2

Expand



flatten

```
List<List<String>> listOfList = [
  ["a", "b", "c"],
  ["d", "e", "f"]
];

// Flattening the list of lists
List<String> singleList = listOfList.expand((list) => list).toList();

// Printing the result
print(singleList); // Output: [a, b, c, d, e, f]
```

expand

Do a map and flatten the results into 1 list

Each book has a list of authors. **expand** combines them to produce a single list of **all** authors

```
List<Book> books = [
  Book("Head First Dart", ["Dawn Griffiths", "David Griffiths"]),
  Book("Dart in Action", ["Dmitry Jemerov", "Svetlana Isakova"]),
];

// Flattening the list of authors
var authors = books.expand((book) => book.authors).toList();
print(authors);
```

Sort a List using Lambda

Sort strings by length (shortest to longest)

```
val names = listOf("Abderahame", "Abdelkarim", "Ali", "Sarah", "Samira", "Farida")
println(">Sorted by length:")
var sorted = names.sortedBy { it.length }
println(sorted)

println("\n>Sorted by length and then alphabetically:")
//Sort strings by length (shortest, longest) and then alphabetically
sorted = names.sortedWith( compareBy( { it.length }, { it }) )
println(sorted)
```

Use .compareBy for multi-step comparisons

```
class Person(val name: String,
             val age: Int)
```

```
fun sortPersons(persons: List<Person>) =
    persons.sortedWith(
        compareBy(Person::name,
                  Person::age))
```

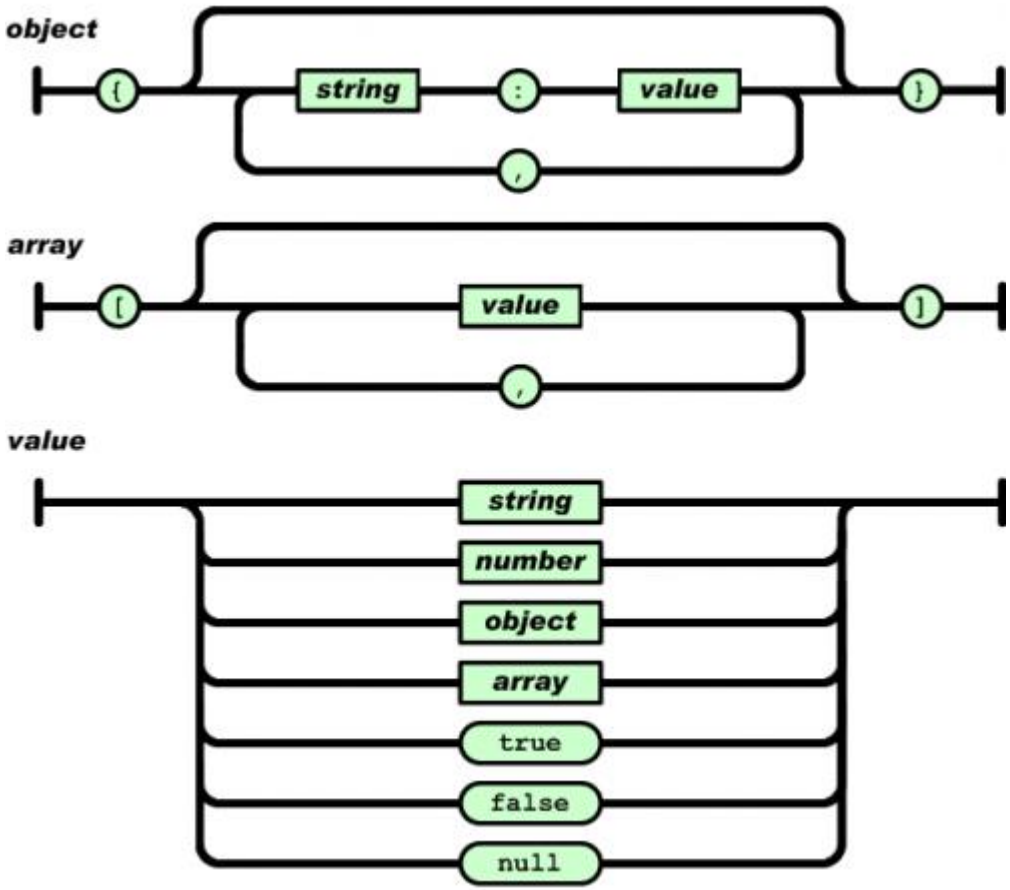
Use .apply for object initialization or for changing multiple attributes

```kotlin
val conference = Conference( name: "Kotlin Conf.",  city: "Istanbul",  fee: 300.0)

// Version 1 👎 - Change the conference city and fee then print it
conference.city = "Doha"
conference.fee = 200.0
println(conference)

// Version 2 ** Best 👍 ** - Change the conference city and fee then print it
// .apply changes the object and returns it
// .also execute some processing on the object and returns it
conference.apply { this: Conference
    city = "Doha"
    fee = 200.0
}.also { println(it) }
```

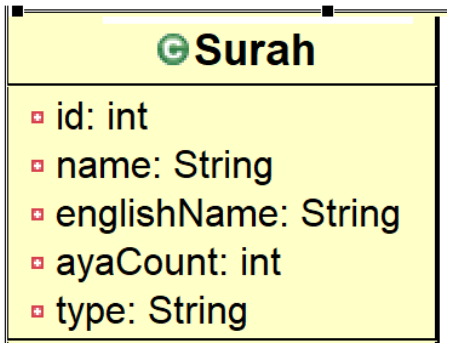

{;}
JSON


object
{
string
:
value
}
,
array
[
value
]
,
value
string
number
object
array
true
false
null

JSON Data Format

- **JSON** (JavaScript Object Notation) is a very popular lightweight data format to transform an object to a **text** form to ease storing and transporting data
- **Json** class could be used to transform an object to json or transform a json string to an object

Transform an instance of Surah class to a JSON string:




```
val fatiha = Surah(1, "الفاتحة", "Al-Fatiha", 7, "Meccan")
val surahJson = Json.encodeToString(fatiha)

// Converting a json string to an object
val surah = Json.decodeFromString<Surah>(surahJson)
```

```
{    "id": 1,
     "name": ","الفاتحة
     "englishName": "Al-Fatiha",
     "ayaCount": 7,
     "type": "Meccan"
 }
```

@Serializable

- To use Json sterilization the class must be annotated with **@Serializable**

```
@Serializable
data class Surah (
    val id : Int,
    val name: String,
    val englishName : String,
    val ayaCount : Int,
    val type: String
)
```

Read JSON file

- Read a JSON file and convert its content to objects

```
val filePath = "data/surahs.json"
val fileContent = File(filePath).readText()

val surahs = Json.decodeFromString<List<Surah>>(fileContent)
```

- To utilize the **`@Serializable`** and **`Json`** class functionalities, ensure that you include the required dependencies in the `build.gradle` file of the module, as detailed in the documentation page.

You may use https://plugins.jetbrains.com/plugin/10054-generate-kotlin-data-classes-from-json Android Studio plugin to generate a Kotlin class from a json string!

Summary

- To start thinking in the functional style ***avoid loops*** and instead use Lambdas
  - Widely used for list processing and GUI building to handle events
- A list can be processed in a pipeline
  - Typical pipeline operations are filter, map and reduce
- JSON is a very popular lightweight data format to **transform an object to a text form** to ease storing and transporting data