# Сериализация

Николай Пономарев

# Что это и зачем нужно

**Сериализация** — это преобразования объектов и структур данных в известный формат с возможностью последующего восстановления. Процесс обратный сериализации называется **десериализация**. Основные применения:

- сохранение данных в локальное хранилище или базу данных
- передача данных, например по сети

# kotlinx.serialization

Для Kotlin существует официальная библиотека `kotlinx.serialization`:

- официальный GitHub: `https://github.com/Kotlin/kotlinx.serialization`
- краткий гайд на *официальном сайте Kotlin*
- подробная документация на *GitHub*

Кстати, буква x в названии пакета образована от слова `eXtension`. В этот пакет попадают официальные библиотеки, которые не включены в `stdlib`. Например:

- `kotlinx.coroutines` – корутины
- `kotlinx.datetime` – работа с временем
- `kotlinx.cli` – парсер для CLI

# Форматы

Официально библиотека поддерживает:

- JSON
- CBOR
- Protocol Buffers
- Java .properties
- HOCON

Благодаря сообществу так же поддерживаются:

- TOML
- XML
- YAML
- множество других

# JSON

- JSON (JavaScript Object Notation) – один из самых популярных и человеко-читаемых форматов. Несмотря на то, что он произошел от JavaScript может использоваться в сочетании с любым языком программирования.
- В `kotlinx.serialization` только JSON считается стабильным, поддержка остальных форматов пока что экспериментальна.
- Примеры использования: конфигурационные файлы Windows Terminal и VS Code.
- Возможно использование схем (schema) – описания кодируемых данных.

# JSON. Типы

JSON поддерживает следующие типы данных:

- `null`
- Number – число, целые и вещественные числа не различимы, максимального числа так же нет
- String – строка, записывается в двойных кавычках, поддерживает экранирование символов через \
- Boolean – принимает `true` или `false`
- Array – упорядоченный список с элементами любого типа, записывается через [], разделяется запятыми
- Object – набор вида ключ-значение, где ключ имеет тип String, записывается через {}, пары отделяются запятыми, а ключ и значение разделяются двоеточием

# JSON. Пример

```json
{
    "firstName": "John",
    "lastName": "Smith",
    "isAlive": true,
    "age": 27,
    "address": {
        "city": "New York",
        "postalCode": "10021-3100"
    },
    "children": [
        "Leah",
        "Mikey"
    ],
    "spouse": null
}
```

# JSON Lines

Формат – альтернатива CSV, каждая строка представляет собой валидный JSON объект или массив.

```
[   "Name", "Session", "Score", "Completed"]
["Gilbert",    "2013",      24,        true]
[  "Alexa",    "2013",      29,        true]
[    "May",   "2012B",      14,       false]
["Deloise",   "2012A",      19,        true]
```

или

```
{"name": "Gilbert", "wins": [["straight", "7♠"], ["one pair", "10♥"]]}
{"name": "Alexa", "wins": [["two pair", "4♠"], ["two pair", "9♠"]]}
{"name": "May", "wins": []}
{"name": "Deloise", "wins": [["three of a kind", "5♠"]]}
```

# Protocol Buffers

- Protocol Buffers – бинарный формат сериализации, предложенный Google в качестве замены XML.
- Для упрощения жизни разработчиков, Protobuf требует наличие схемы.
- По умолчанию ни одно поле не может быть `null`.
- Продвигается Google для использования в Android.

# Protobuf. Пример

```
syntax = "proto3";

message Person {
    message Address {
        string city = 1;
        string postalCode = 2;
    }
    string firstName = 1;
    string lastName = 2;
    bool isAlive = 3;
    uint32 age = 4;
    Address address = 5;
    repeated string children = 6;
    string spouse = 7;
}
```

# TOML

TOML – минималистичный формат, похожий на .ini файлы, но с более строгой спецификацией.

```toml
firstName = "John"
lastName = "Smith"
isAlive = true
age = 27
children = [ "Leah", "Mikey" ]

[address]
city = "New York"
postalCode = "10021-3100"
```

# YAML

YAML – минималистичный формат. Является надмножеством JSON. Используется, например, для настройки GitHub Actions.

```yaml
---
firstName: John
lastName: Smith
isAlive: true
age: 27
address:
  city: New York
  postalCode: 10021-3100
children:
  - Leah
  - Mikey
spouse:
```

# Простой пример. Сериализация

```
import kotlinx.serialization.*
import kotlinx.serialization.json.*

@Serializable
data class Project(val name: String, val language: String)

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")
    println(Json.encodeToString(data))
}
```

Вывод:

```
{
    "name": "kotlinx.serialization",
    "language": "Kotlin"
}
```

# Простой пример. Десериализация

```
@Serializable
data class Project(val name: String, val language: String)

fun main() {
    val data = Json.decodeFromString<Project>(
        """{
            "name": "kotlinx.serialization",
            "language": "Kotlin"
        }""")
    println(data)
}
```

Вывод:

```
Project(name=kotlinx.serialization, language=Kotlin)
```

## Что сериализуется

По-умолчанию сериализуются все свойства класса, но с некоторыми особенностями:

- Сериализуются только свойства имеющие backing field и не делегаты
- В первичном конструкторе класса могут быть только свойства
- Свойства с значением по-умолчанию не будут сериализоваться
- Если свойство не нужно сериализовать, его можно обозначить аннотацией `@Transient`, у такого свойства должно быть значение по умолчанию
- Если свойство – это ссылка, то объект по ссылке тоже должен иметь аннотацию `@Serializable`
- Типы generic'ов выводятся во время компиляции
- Можно менять название свойств при сериализации с помощью `@SerialName(name: String)`

## Сложный пример

```
@Serializable
data class Child(val name: String)

@Serializable
data class Address(val city: String = "London", val postalCode: String)

@Serializable
data class Person(
    @SerialName("firstName") val name: String,
    @SerialName("lastName") val surname: String,
    val isAlive: Boolean = true,
    val age: Int,
    val address: Address,
    val children: List<Child>,
    val spouse: String?
)
```

# Сложный пример

```
fun main() {
    val employee = Person(
        name = "John",
        surname = "Smith",
        age = 27,
        address = Address(city = "New York", postalCode = "10021-3100"),
        children = listOf(Child("Leah"), Child("Mikey")),
        spouse = null
    )
    val json = Json { prettyPrint = true }
    println(json.encodeToString(employee))
}
```

# Сложный пример. Вывод

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 27,
    "address": {
        "city": "New York",
        "postalCode": "10021-3100"
    },
    "children": [
        {
            "name": "Leah"
        },
        {
            "name": "Mikey"
        }
    ],
    "spouse": null
}
```

# Что еще можно делать

- Писать собственные сериализаторы
  - Простой пример: хотим закодировать зеленый цвет `0x00ff00` читаемо – не числом. Возможные варианты: строка `"00ff00"`, массив `[0,255,0]`, JSON объект `{"r":0,"g":255,"b":0}` – все это возможно
- Делать внешние классы сериализуемыми
- «Перегружать» сериализаторы с помощью `@Contextual`
- Работать с сложной иерархией классов
- Настраивать вывод: каждый формат имеет собственные настройки