# Welcome to Serialization

Encoding, serialization, marshalling
Unencoding, parsing, deserialization, unmarshalling

## Plan

- 10 minutes - Introduction
- 20 minutes - Workshop
- 30 minutes - Presentation material
- 10 minutes - Q&A

## Sources

- Designing Data-Intensive Applications by Martin Kleppmann - CHAPTER 4 Encoding and Evolution
- Thinking in Java by Bruce Eckel - Object serialization

## What is serialization?

What task does it solve?

The translation from the in-memory representation to a byte sequence is called encoding (also known as serialization or marshalling), and the reverse is called decoding (parsing, deserialization, unmarshalling).

⚠ IT HAS Nothing common with serializable level in database transactions

## Workshop

Write your own encoder/decoder for class

```kotlin
data class Customer(val id: Long, val name: String, val balanceAmount: BigDecimal)
```

### Sample implementation

```kotlin
fun encode(customer: Customer): String {
    return "${CUSTOMER_OBJECT_PREFIX}${customer.id};${customer.name};${customer.balanceAmount}"
}

fun decode(encodedString: String): Customer {
    if (!encodedString.startsWith(CUSTOMER_OBJECT_PREFIX)) {
        throw RuntimeException("Cannot parse customer - it's not a customer")
    }
    val withoutPrefix = encodedString.removePrefix(CUSTOMER_OBJECT_PREFIX)
    val fields = withoutPrefix.split(";")
    return Customer(
        id = fields[0].toLong(),
        name = fields[1],
        balanceAmount = BigDecimal(fields[2])
    )
}
```

# Inner objects workshop

Write your own encoder/decoder for class Customer

```
data class BankCustomer(val id: Long, val name: String, val bankAccount: BankAccount)
data class BankAccount(val regionId: Long, val balanceAmount: BigDecimal)
```

## Sample implementation

```
fun encode(customer: BankCustomer) = with(customer) {
    "${CUSTOMER_OBJECT_PREFIX}${id};${name};${BankAccountSerializer.encode(bankAccount)}"
}

fun decode(encodedString: String): BankCustomer {
    if (!encodedString.startsWith(CUSTOMER_OBJECT_PREFIX)) {
        throw RuntimeException("Cannot parse bank customer - it's not a bank customer")
    }
    val withoutPrefix = encodedString.removePrefix(CUSTOMER_OBJECT_PREFIX)
    val fields = withoutPrefix.split(";")
    return BankCustomer(
        id = fields[0].toLong(),
        name = fields[1],
        bankAccount = BankAccountSerializer.decode(fields[2] + fields[3])
    )
}
```

# Java example

```java
public record BankAccountJava(long regionId, BigDecimal balanceAmount) implements Serializable {
    @Serial
    private static final long serialVersionUID = 1L;
}
```

```java
public record BankCustomerJava(long id, String name, BankAccountJava bankAccount) implements Serializable { //....
```

## Encoding and decoding

```java
// Create objects
BankAccountJava bankAccount = new BankAccountJava(1L, new BigDecimal("1000.50"));
BankCustomerJava customer = new BankCustomerJava(1L, "John Doe", bankAccount);
// Encoding
ByteArrayOutputStream byteArrayOut = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(byteArrayOut);
out.writeObject(customer);
// Decoding
byte[] serializedData = byteArrayOut.toByteArray();
ByteArrayInputStream byteArrayIn = new ByteArrayInputStream(serializedData);
ObjectInputStream in = new ObjectInputStream(byteArrayIn);
BankCustomerJava deserializedCustomer = (BankCustomerJava) in.readObject();
```

## Formats

- Language-specific formats (java.io.serializable, Ruby — marshal, Python — pickle etc.)
- Text encodings - Json, XML, CSV
- Binary encodings — Thrift, Protocol buffers, Avro

## Problems

- Reading by machine
- Reading by human
- Types — Big decimal, or file as field
- Schema as common protocol
- Versioning

## Java serializable

- Language specific
- No human readable text
- Types — any, but problems with non implementing serializable
- Schema — as java file
- Versioning — no backward compatibility - serialVersionUID

## JSON/XML

- On any language
- Human readable
- Types — can be limited (Json — difficulties with binary types)
- Schema — exists, no central communication
- Versioning — custom

## CSV

- No standard based — any languages
- Human readable
- No types — string — based
- No schema
- No versioning - custom

## Protobuf, Avro, Thirft

Cross-platform binary formats

- On many languages
- Not human readable — need translator
- Types — can be limited
- Schema — exists, no central communication
- Versioning — in the protocol

Difference

- Protobuff - gRPC
- Avro - General serialization (Kafka or BigData table)
- Thrift - RPC (Facebook), Cassandra

## Avro schema file

```
{
  "type": "record",
  "name": "UserSampleAvro",
  "namespace": "com.walkingfeet.example.avro",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "age",  "type": "int" },
    { "name": "email", "type": ["null", "string"], "default": null }
  ]
}
```

## Avro deserialisation

```
// Encoding
val user = UserSampleAvro("Dmitry", 31, "dmitry@example.com")

val outputStream = ByteArrayOutputStream()
val userWriter: DatumWriter<UserSampleAvro> = SpecificDatumWriter(UserSampleAvro::class.java)

val dataFileWriter: DataFileWriter<UserSampleAvro> = DataFileWriter(userWriter)

dataFileWriter.create(user.schema, outputStream)
dataFileWriter.append(user)
dataFileWriter.close()

//Decoding
val inputStream = ByteArrayInputStream(outputStream.toByteArray())
val userReader: DatumReader<UserSampleAvro> = SpecificDatumReader(UserSampleAvro::class.java)
val dataFileReader: DataFileStream<UserSampleAvro> = DataFileStream(inputStream, userReader)

val deserializedUser = mutableListOf<UserSampleAvro>()
while (dataFileReader.hasNext()) {
    val readUser: UserSampleAvro = dataFileReader.next()
    deserializedUser.add(readUser)
}
dataFileReader.close()
```

# Q&A

Powered by Slidev