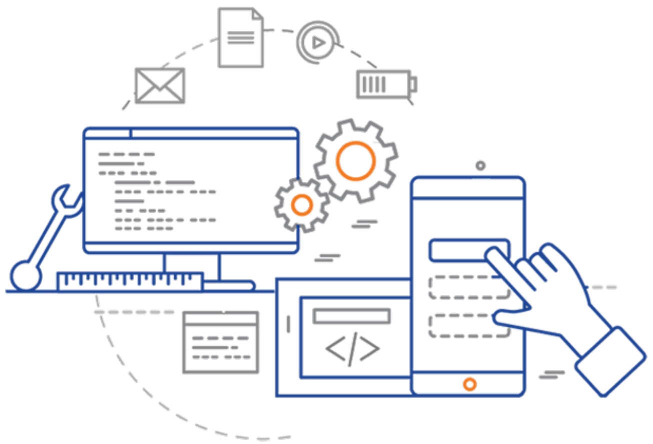


CS 398: Application Development

Working with Data

Data modelling; data formats; serialization.

Managing Data

Most software operates on data. This might be a primary function of your application (e.g. image editor) or a secondary function (e.g. user preferences).

Some examples of data that you might need to store:

- The data that you are manipulating (e.g. an image from an image editor).
- The position and size of the application window, or user preferences.
- The public key to connect to a remote machine.
- The software license key for your application.

Operations you may need to do:

- Store data to a disk, or retrieve it from disk into some useful data structure.
- Transmit it to another process, possibly running on a different machine.

Working with Data

Data Types

A type is a way of categorizing our data so that the computer knows how we intend to use it.

Different kinds of types:

- **Primitive Types**: these are intrinsically understood by a compiler, and will include boolean, and numeric data types. e.g. boolean, integer, float, double.
- **Strings**: Any text representation which can include characters (char) or longer collections of characters (string). Strings are complicated to store and manipulate.
- **Compound data types**: A combination of primitive types. e.g. an Array of Integers.
- **Pointers**: A data type whose value points to a location in memory. The underlying data may resemble a long Integer, but they are treated as a separate type to help protect our programs.
- **Abstract data type**: We treat ADTs as different because they describe a structure, and do not actually hold any concrete data. We have a template for an ADT (e.g. class) and concrete realizations (e.g. objects). These can be singular, or stored as Compound types as well.

Types

We declare variables in a programming language using the types for that language. For example, in Kotlin, we can assign the value 4 to different variables, each representing a different type.

```
val a:Int = 4
val b:Double = 4.0
val c:String = "4"
```

By using different types, we’ve made it clear to the compiler that a and b do not represent the same value.

```
>>> a==b
error: operator '==' cannot be applied to 'Int' and 'Double'
```

Types

```
>>> c.length
res13: kotlin.Int = 1

>>> a.length
error: unresolved reference: length
a.length
  ^


>>> b/4
res15: kotlin.Double = 1.0

>>> c/4
error: unresolved reference. None of the following candidates is applicable because of
receiver type mismatch:
public inline operator fun BigDecimal.div(other: BigDecimal): BigDecimal defined in kotlin
public inline operator fun BigInteger.div(other: BigInteger): BigInteger defined in kotlin
c/4
 ^
```

There are also behaviours and properties specific to each type. You cannot fetch the length of an Int, or divide a String by 4.

Storing Data

All this is fairly obvious. Why is it important?

Types constrain and limit how we represent data.

All data in memory needs to be stored in a series of primitives, or strings, or a collection, or an instance of an ADT, or a stream of binary data. The challenge is finding the appropriate representation for the data that you need to store.

Data can be simple, consisting of a single field (e.g. the user's name), or complex, consisting of a number of related fields (e.g. a customer with a name, address, job title). Typically we group related data into classes, and objects representing instances of that class.

```
data class Customer(id:Int, name:String, city:String)
val new_client = Customer(1001, "Jane Bond", "Waterloo")
```

Singular vs. Collections

Data items can be singular (e.g. one particular customer), or part of a collection (e.g. all of my customers). A singular example would be a customer record.

```
val new_client1 = Customer(1001, "Jane Bond", "Waterloo")
val new_client2 = Customer(1002, "Bruce Willis", "Kitchener")
```

A collection example might be bank transactions, where each transaction represents the deposit or withdrawal, the date when it occurred, the amount and so on.

```
data class Tx(id:Int, date:String, amount:Double, currency: String)
val transactions = mutableList()
transactions.add(Tx(1001, "2020-06-06T14:35:44", "78.22", "CDN"))
transactions.add(Tx(1001, "2020-06-06T14:38:18", "12.10", "USD"))
transactions.add(Tx(1002, "2020-06-06T14:42:51", "44.50", "CDN"))
```

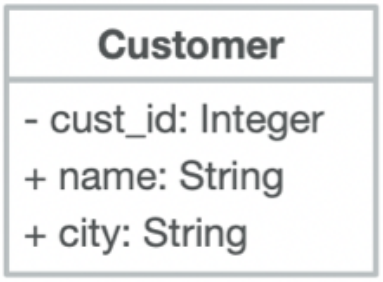
A **data model** is an abstraction of how our data is structured and how data elements relate to one another. We will often have a canonical data model that we can use as at the basis for different representations.

A data model is essential because it provides “ground truth” of how your data should be represented.

We may need to represent this data in different formats throughout our program, but we derive them all from this single “correct” and accurate representation.

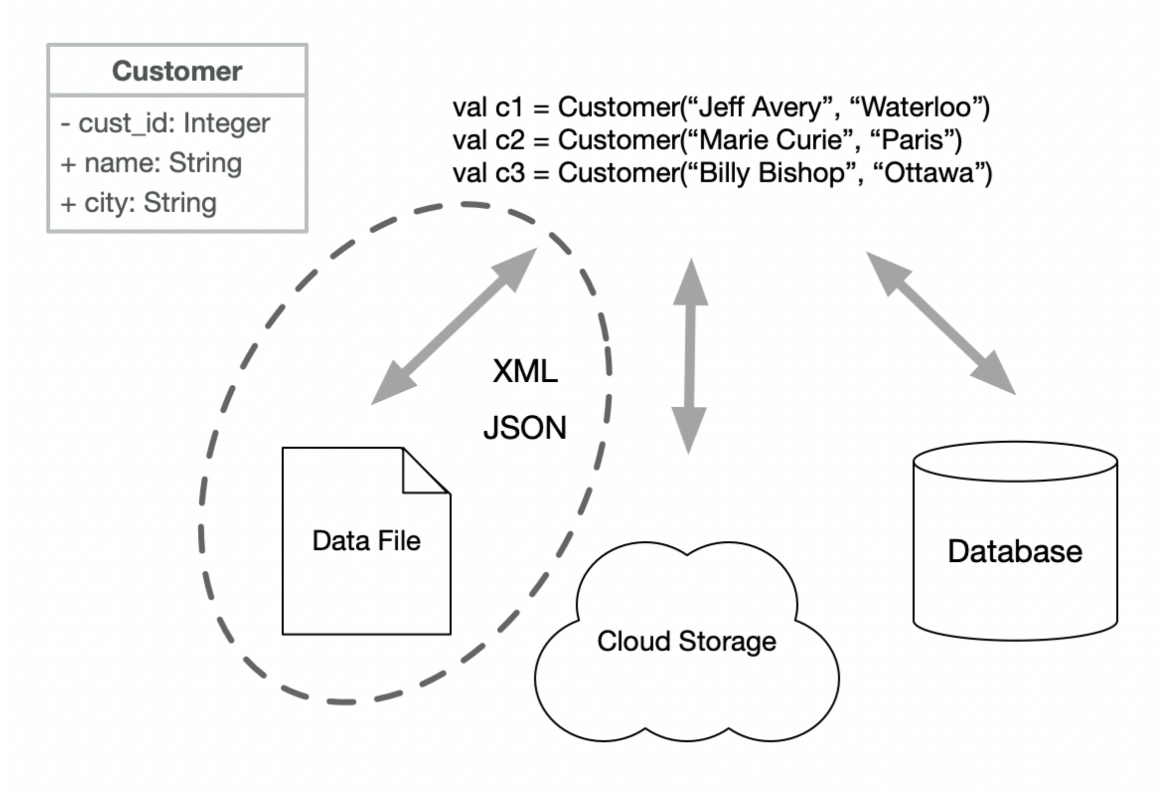
A data model demonstrates how different types combine into a single related concept, “Customer”.

A UML class diagram isn’t a perfect representation of this, since it may contain implementation-specific details, but it’s reasonably close.



Data models are important because the provide the foundation for migrating data between different domains e.g. data files, cloud storage or a database.

Let's start with a data file. What do we need to consider to store our class information in a data file?



```
val c1 = Customer("Jeff Avery", "Waterloo")
val c2 = Customer("Marie Curie", "Paris")
val c3 = Customer("Billy Bishop", "Ottawa")
```

Data Formats

Character Encoding

Let's focus on text, or character data for a moment. This is the most common type of data that we represent externally. Like other data, characters in memory are stored as binary i.e. numeric values in a range. To display them, we need some agreement on which number represents each character.

Our desire to be able to encode all characters in all language is balanced by our need to be efficient. i.e. we want to use the least number of bytes per character that we're storing.

Examples of two different encodings:

1. **US-ASCII** stores each character in 7 bits (range of 0-127). Although it was sufficient for "standard" English language typewriter symbols, this is rarely used anymore since it cannot handle languages other than English.

2. **Unicode** uses 1, 2, or 4 bytes for each character (i.e. it's a variable length, multi-byte format). UTF-8 can only store the first set of characters (that can be addressed by a single byte), UTF-16 and UTF-32 expand this range to include all characters.

Unicode/UTF-8 is considered standard encoding, unless we have a need for multibyte data.

CSV Files

We know that we will use UTF-8, but that only describes how the characters will be stored.

We also need to determine how to structure our data in a way that reflects our data model.

We'll talk about three different data structure formats for managing text data, all of which will work with UTF-8.

Let's consider storing data in a file. The simplest way to store records might be to use a CSV (comma-separated values) file. We use this structure:

- Each row corresponds to one record (i.e. one object)
- The values in the row are the fields separated by commas.

For example, our transaction data file stored in a comma-delimited file would look like this:

```
1001, 2020-06-06T14:35:44, 78.22, CDN
1002, 2020-06-06T14:38:18, 12.10, USD
1003, 2020-06-06T14:42:51, 44.50, CDN
```

CSV is literally the *simplest possible thing* that we can do, and *sometimes* it’s good enough.

It has some **advantages**:

- its extremely easy to work with, since you can write a class to read/write it.
- it’s human readable which makes testing/debugging much easier.
- it’s pretty space efficient.

However, this comes with some pretty big **disadvantages** too:

- It also doesn’t work very well if your data contains a delimiter (e.g. a comma).
- It assumes a fixed structure and doesn’t handle variable length records.
- It doesn’t work very for complex or multi-dimensional data. e.g. a Customer class.

```
// how do you store this as CSV? the list is variable length
data class Customer(id:Int, name:String, transactions:List<Transactions>)
```

Data streams are used to provide support for reading and writing “primitives“ from streams.

```
var file = FileOutputStream("hello.txt")
var stream = DataOutputStream(file)

stream.writeInt(100)
stream.writeFloat(2.3f)
stream.writeChar('c')
stream.writeBoolean(true)
```

We can use streams to write our transaction data to a file quite easily.

```
val filename = "transactions.txt"
val delimiter = "," // comma-delimited values

// add a new record to the data file
fun append(txID:Int, amount:Float, curr:String = "CDN") {
  val datetime = LocalDateTime.now()
  File(filename).appendText("$txID $delimiter $datetime $delimiter $amount\n", Charsets.UTF_8)
}
```

XML Files

Extensible Markup Language (XML) is a markup language that designed for data storage and transmission.

Defined by the World Wide Web Consortium's XML specification, it was the first major standard for markup languages. It's structurally similar to HTML[9], with a focus on data transmission (vs. presentation).

- Structure consists of pairs of tags that enclose data elements.

```
<name>Jeff</name>
```

- Attributes can extend an element

```
<img src=”madonna.jpg”></img>
```

Example of a music collection structured in XML[1].

```
<catalog>
    <album>
        <title>Empire Burlesque</title>
        <artist>Bob Dylan</artist>
        <country>USA</country>
        <company>Columbia</company>
        <price>10.90</price>
        <year>1985</year>
    </album>
    <album>
        <title>Innervisions</title>
        <artist>Stevie Wonder</artist>
        <country>US</country>
        <company>The Record Plant</company>
        <price>9.90</price>
        <year>1973</year>
    </album>
</catalog>
```

Album is a record, containing Fields for title, artist etc.

Note the opening and closing tags. If XML looks like HTML, that's because they're both descended from a common ancestor language, SGML.

—

1. If you don't know these albums, you should look them up. Steve Wonder is a musical genius. Dylan is, well, *Dylan*.

JSON Files

JavaScript Object Notation (JSON) is an open standard file format, and data interchange format that's commonly used on the web.

JSON consists of attribute:value pairs and array data types. It's based on JavaScript object notation, but is language independent. It was standardized in 2013 as ECMA-404.

JSON has become increasingly popular due to its simpler syntax compared to XML.

- Data elements consist of name/value pairs
- Fields are separated by commas
- Curly braces hold objects
- Square brackets hold arrays

Here’s the music collection in JSON.

```
{ "catalog": {
  "albums": [
    {
      "title":"Empire Burlesque",
      “artist":"Bob Dylan",
      "country":"USA",
      "company":"Columbia",
      "price":"10.90",
      "year":"1988"
    }, {
      "title":"Innervision",
      “artist":"Stevie Wonder",
      “country":"US",
      “company":"The Record Plant”,
      "price":"9.90",
      "year":"1973"
    }
  ]
}}
```

Album is a record, containing Fields for title, artist etc.

No tags, just keys and values! Much easier to read, since it’s all meaningful data.

Condensing closing tags makes JSON easier to read.

```
{ “employees":[
  { "first":"John", "last":"Zhang", “dept":"Sales"},
  { "first":"Anna", "last":"Smith", "dept":"Engineering"}
]}
```

Compare this to the corresponding XML:

```
<employees>
  <employee><first>John</first> <last>Zhang</last> <dept>Sales</dept></employee>
  <employee><first>Anna</first> <last>Smith</last> <dept>Engineering</dept></employee>
</employees>
```

…and this is a *small* record.

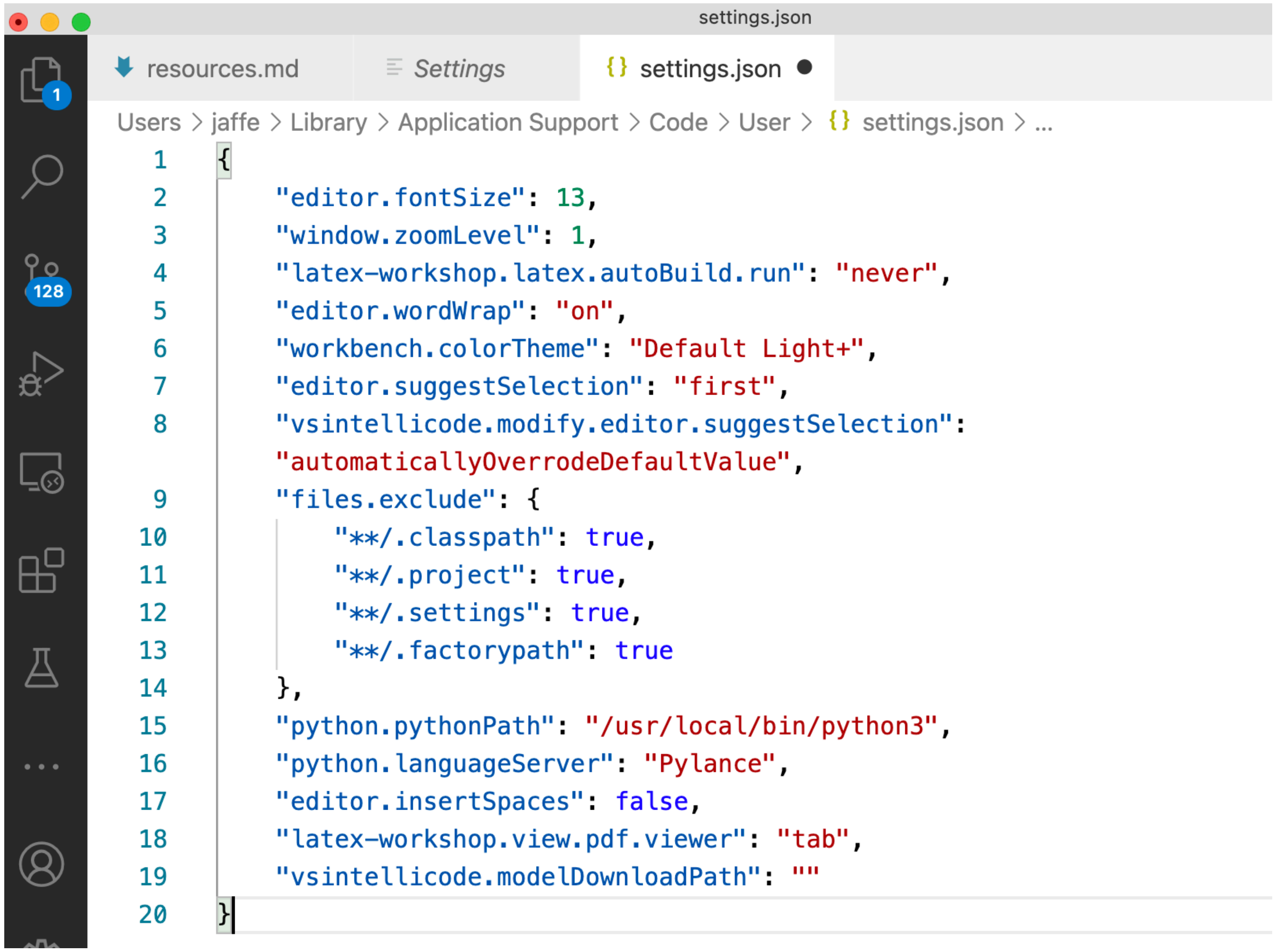
JSON also handles arrays better.

Array in XML:

```
<name>Celia</name>
<age>30</age>
<cars>
  <model>Ford</model>
  <model>BMW</model>
  <model>Fiat</model>
</cars>
```

Array in JSON:

```
{
"name":"Celia",
"age":30,
"cars":[ "Ford", "BMW", "Fiat" ]
}
```

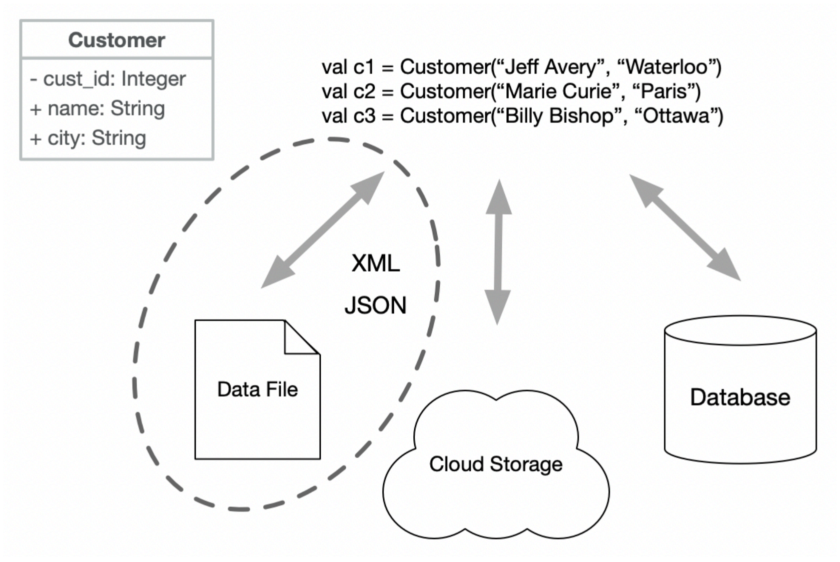

settings.json
resources.md
Settings
settings.json
Users > jaffe > Library > Application Support > Code > User > settings.json > ...

```json
{
    "editor.fontSize": 13,
    "window.zoomLevel": 1,
    "latex-workshop.latex.autoBuild.run": "never",
    "editor.wordWrap": "on",
    "workbench.colorTheme": "Default Light+",
    "editor.suggestSelection": "first",
    "vsintellicode.modify.editor.suggestSelection":
    "automaticallyOverrodeDefaultValue",
    "files.exclude": {
        "**/.classpath": true,
        "**/.project": true,
        "**/.settings": true,
        "**/.factorypath": true
    },
    "python.pythonPath": "/usr/local/bin/python3",
    "python.languageServer": "Pylance",
    "editor.insertSpaces": false,
    "latex-workshop.view.pdf.viewer": "tab",
    "vsintellicode.modelDownloadPath": ""
}
```

Visual Studio Code stores user settings as JSON. It's clean enough to read that users can edit this file by-hand to change preferences.

So, our application data resides in data structures, in memory. How do we make use of JSON or XML?

- In our code, we convert objects into XML or JSON. We then save the raw XML or JSON in a data file.
- We could also write functions to instantiate objects from raw XML or JSON.


Customer
- cust_id: Integer
+ name: String
+ city: String
val c1 = Customer("Jeff Avery", "Waterloo")
val c2 = Customer("Marie Curie", "Paris")
val c3 = Customer("Billy Bishop", "Ottawa")
XML
JSON
Data File
Cloud Storage
Database

XML and JSON are commonly used to store data in files.

Serialization

Java introduced the idea of serialization: a built-in mechanism to convert an object to a stream that could be transmitted, or persisted.

- **Serialization**: save your object to a file.
- **Deserialization**: instantiate an object from your previously saved file.

We can use this to convert objects to and from JSON, which we can then save, print, etc. as we wish.

```
plugins {
   id 'org.jetbrains.kotlin.plugin.serialization' version '1.6.10'
}

dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.2"
}
```

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

@Serializable
data class Project(
   val name: String,
   val owner: Account,
   val group: String = "R&D"
)

@Serializable
data class Account(val userName: String)

val moonshot = Project("Moonshot", Account("Jane"))
val cleanup = Project("Cleanup", Account("Mike"), "Maint")

fun main() {
   val string = Json.encodeToString(listOf(moonshot, cleanup))
   println(string)
   // [{“name":"Moonshot","owner":{"userName":"Jane"}},{"name":"Cleanup","owner" {"userName":"Mike"},"group":"Maint"}]

   val projectCollection = Json.decodeFromString<List<Project>>(string)
   println(projectCollection)
   // [Project(name=Moonshot, owner=Account(userName=Jane), group=R&D), Project(name=Cleanup, owner=Account(userName=Mike), group=Maint)]
}
```

Classes need to be @Serializable to convert to JSON.

JSON is just a string, so we can save it directly to a file - just as we did with CSV data.

Objects -> JSON

```
@Serializable
data class Project(
    val name: String,
    val owner: Account,
    val group: String = "R&D"
)

@Serializable
data class Account(val userName: String)

val moonshot = Project("Moonshot", Account("Jane"))
val cleanup = Project("Cleanup", Account("Mike"), “Maint")

val string = Json.encodeToString(listOf(moonshot, cleanup))
val projectCollection = Json.decodeFromString<List<Project>>(string)
```

String:

```
[
  { “name”:”Moonshot",
    “owner":{"userName":"Jane"}
  },
  { “name":"Cleanup",
    “owner":{"userName":"Mike"},
    “group”:"Maint"
  }
]
```

The JSON that is produced is well-formed, and will allow us to recreate objects.
Fields will be mapped directly to the fields in the class constructor.

JSON -> Objects

```
@Serializable
data class Project(
    val name: String,
    val owner: Account,
    val group: String = "R&D"
)

@Serializable
data class Account(val userName: String)

val moonshot = Project("Moonshot", Account("Jane"))
val cleanup = Project("Cleanup", Account("Mike"), “Maint")

val string = Json.encodeToString(listOf(moonshot, cleanup))
val projectCollection = Json.decodeFromString<List<Project>>(string) ★
```

String:

```
[
  { “name”:”Moonshot",
    “owner":{"userName":"Jane"}
  },
  { “name":"Cleanup",
    “owner":{"userName":"Mike"},
    “group”:"Maint"
  }
]
```

The decoder relies on strict formatting, and a matching constructor.
This is why you should be very careful when manually constructing your JSON!

Binary Data

These examples have all talked about reading/writing text data. What if I want to process binary data? Many binary data formats (e.g. JPEG) are defined by a standard, and will have library support for reading and writing them directly.

Kotlin also includes object-streams that support reading and writing binary data, including entire objects. You can, for instance, save an object and it's state (serializing it) and then later load and restore it into memory (deserializing it).

```
class Emp(var name: String, var id:Int) : Serializable {}
var file = FileOutputStream(“datafile")
var stream = ObjectOutputStream(file)

var ann = Emp(1001, "Anne Hathaway", "New York")
stream.writeObject(ann)
```