Data Representation

CS 346 Application Development

Applications = Data + Computation

Meaningful applications interact with data.

- e.g., text editors write text files; graphics editors load and modify images.

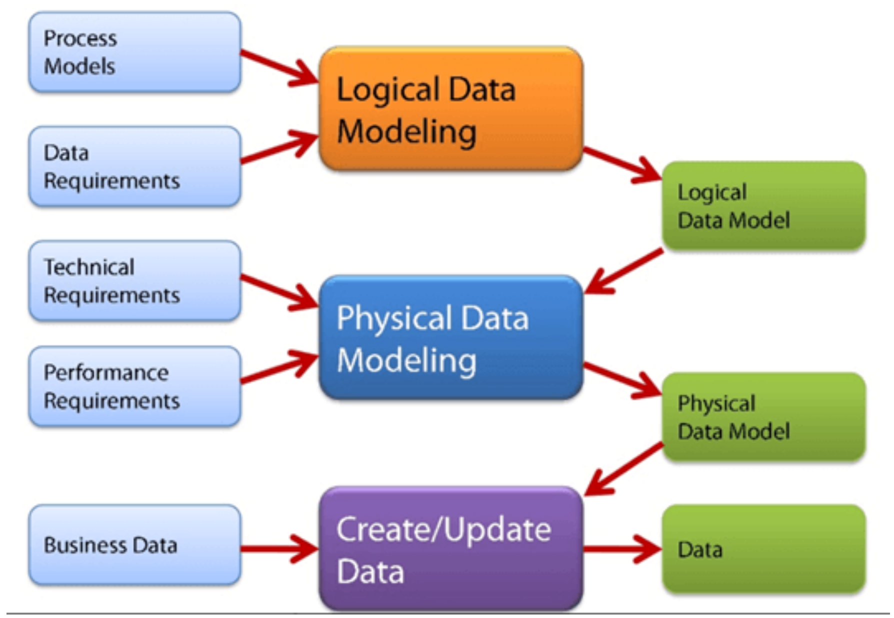
Consider what data your application manipulates:

- What is the source of data: is it loaded from a file or streamed from a service?
- Does it have a format that I need to be able to manipulate? e.g., JPEG images.
- Do I need to cache data locally, or do I reload it as-needed? If so, when and how?
- How do I represent this data in my code?

- You also need to consider how the data will be used:
  - Is the data specific to a user (e.g., password), or application (e.g., window size).
  - Do you need to export or transmit the data, or store is in a shared location?
  - What are the privacy and security implications of transmitting or storing this data?

Data Models

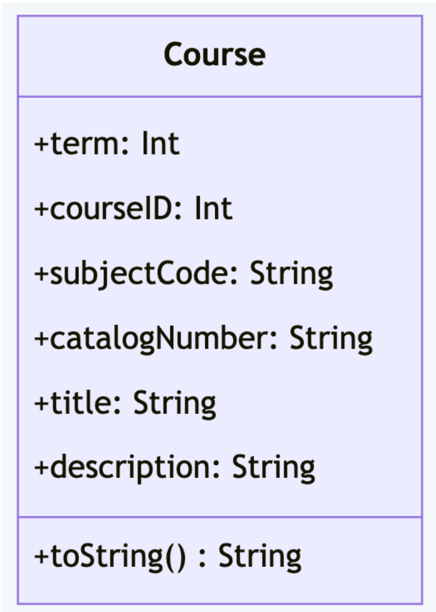
We normally use **data models** to describe our data. These are high-level visual representations that describes elements, their structure and relationships to one another.



Logical Model – rules governing how data is represented (e.g., UML, classes)

Physical model – how system is implemented (e.g., data structures)

Conceptual models – high-level structure (e.g., description)



Class representation (UML)

1251, 01687, “CS”, “346”, “Application Development”, “Introduction to full-stack application design”

Logical records (text representation)

```
val record = Course (
        1251,
        01687,
        “CS”,
        “346”, “Application Development”,
        “Introduction to full-stack application design”
)
```

Logical records (object representation)

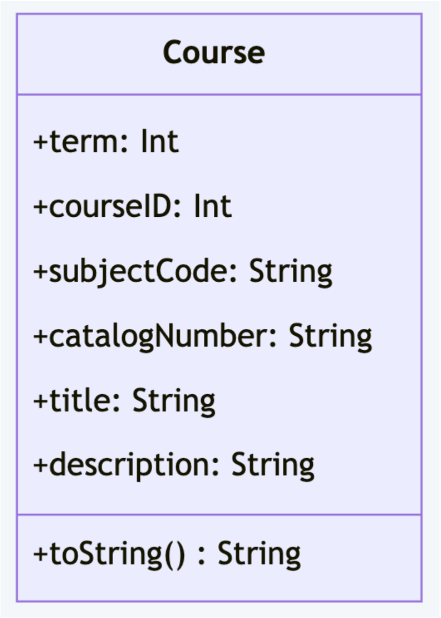
Our data model needs to provide us with some flexibility!
We may need to save to a file, send to a remote machine, save to a database.
Our goals are (1) correct representation, and (2) the ability to adapt to different needs.

Data Classes

The obvious choice for storing records would be to use a data class for the record, and a collection to store a set of records.

```
data class Course (
    val term: Int,
    val courseID: String,
    val subjectCode: String,
    val catalogNumber: String,
    val title: String = “”,
    val description: String = “”
)

val courses = List<Course>() // List of courses
courses.add(Course(1251, ”01687", ”CS”, “346”))
```



Class representation

```kotlin
open class CourseDao(
    val courseId: String,
    val courseOfferNumber: Int,
    val termCode: Int,
    val termName: String,
    val associatedAcademicCareer: String,
    val associatedAcademicGroupCode: String,
    val associatedAcademicOrgCode: String,
    val subjectCode: String,
    val catalogNumber: String,
    val title: String,
    val descriptionAbbreviated: String,
    val description: String,
    val gradingBasis: String,
    val courseComponentCode: String,
    val enrollConsentCode: String,
    val enrollConsentDescription: String,
    val dropConsentCode: String,
    val dropConsentDescription: String,
    val requirementsDescription: String?
)
```

API

```kotlin
open class Course(
    val term: Int,
    val courseID: String,
    val subject: String,
    val catalogNumber: String,
    val title: String,
    val description: String
)
```

DB

Your data may be structured differently across systems. For example, a service may return more fields in a record than you care about internally!

Class conversion

```kotlin
open class Course(
    val term: Int,
    val courseID: String,
    val subject: String,
    val catalogNumber: String,
    val title: String,
    val description: String
) {
     constructor(course: CourseDao): this(
        term = course.termCode,
        courseID = course.courseId,
        subject = course.subjectCode,
        catalogNumber = course.catalogNumber,
        title = course.title,
        description = course.description,
     )
}
```

The courses demo contains three different representations of course data: DB, web service and data class. It has multiple constructors like this.

Managing Data Classes

So, internally we store data in classes.

How do you take an object and:

- write it to a file,
- transmit it over a network connection,
- save it to a database?

ORM vs. manually mapping
What formats are suitable for these scenarios?

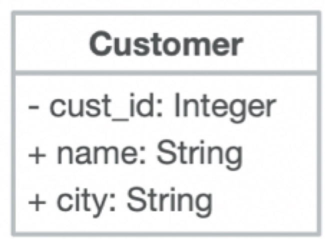
Data formats

What we want in a storage format.

Structured CSV

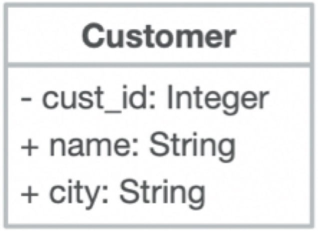
- The simplest way to store records might be to use a CSV (comma-separated values) file. We use this structure:
  - Each row corresponds to one record (i.e. one object)
  - The values in the row are the field for each record, separated by commas.
- For example, transaction data file stored in a comma-delimited file:

```
1001, Jeff Avery, Cambridge
1002, Allison Barnett, Waterloo
1003, John McAfee, Delphi
```



CSV realization of this class data.

Reading/Writing Objects to CSV




```
data class Customer (val cust_id: Int, val name: String, val city: String)

val customers = List<Customer>() // list of customers
customers.add(Customer(1001, "John Hall", "New York"))
customers.add(Customer(1002, "Allison Barnett", "Waterloo"))
customers.add(Customer(1003, "John McAfee", "Delphi"))

File(“output.csv").open("w").use {
    it.write("Customer ID, Name, City\n")
    for (customer in customers) {
        it.write("${customer.cust_id},${customer.name},${customer.city}\n")
    }
}
```

Reading the file will require loading a line and splitting at each delimiter (comma).

Pro/Con of CSV files

CSV is literally the *simplest possible thing* that we can do.

- As a file format, it has some **advantages**:
  - Programming languages can easily work with CSV files (they're just text!)
  - It's pretty space efficient.
  - It's human-readable. Kind-of.
- However, CSV comes with some big **disadvantages**:
  - It doesn't work very well if your data contains the delimiter (e.g. a comma in your city field).
  - It assumes a **fixed structure** and doesn't handle variable length records.
  - It's hard to read! **There is no semantic information** to make sense of it. (i.e., there is no simple way to interpret the structure, no schema file format).
  - It doesn't work for complex, multi-dimensional data. e.g. Customer transactions.

Structured Data formats: XML

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, with a focus on data transmission (vs. presentation).

Structure consists of pairs of tags that enclose data elements. Attributes can be added.

```
<name>Jeff</name>
<img src="mosaic.jpg">This is a caption</img>
```

You can have a schema that describes the data structure! You can validate data.

XML Example

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

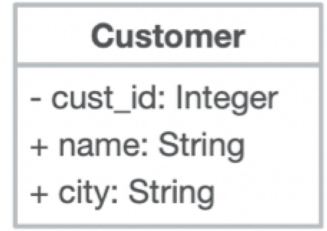
```
<catalog>
    <album>
        <title>Empire Burlesque</title>
        <artist>Bob Dylan</artist>
        <company>Columbia</company>
        <price>10.90</price>
        <year>1985</year>
    </album>
    <album>
        <title>Innervisions</title>
        <artist>Stevie Wonder</artist>
        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

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

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

1. Check out these albums!

Reading/Writing Objects to XML




```
data class Customer (val cust_id: Int, val name: String, val city: String)

val customers = List<Customer>() // list of customers
customers.add(Customer(1001, "John Hall", "New York"))
customers.add(Customer(1002, "Allison Barnett", "Waterloo"))
customers.add(Customer(1003, "John McAfee", "Delphi"))

File("output.xml").open("w").use {
    it.write("<customers>")
    for (customer in customers) {
        it.write("<customer>")
        it.write("<cust_id>${customer.cust_id}</cust_id>")
        it.write("<name>${customer.name}</name>")
        it.write("<city>${customer.city}</city>")
    }
    it.write("</customers>")
}
```

Reading the file will require a very complex parser e.g., Stax for Java.

Pros/Cons of XML

XML provides structure.

- The use of tags, and the optional use of a schema file, means that we can formally define the semantic structure of our data!
- This provides some major advantages compared to CSV.
  - Can rely on structure to infer the meaning of data.
  - You can nest elements e.g., collections of records.

XML is rarely used except in legacy systems. Why?

- Tags "bloat" the data, which results in excessive space requirements.
- Practically impossible to parse without a complex library.

Structured Data Format: YAML

YAML Ain't Markup Language (YAML) is a data serialization language. It's easy for humans to read, and it's commonly used for configuration.

- Three dashes: start of YAML document
- Key: value pairs
- Lists: dash for each element

- Thoughts on human-readable formats
  - Used extensively for **config files**.
  - Indentation used for structure; difficult to parse manually.
  - Not as widely supported as other formats :/

Example from https://www.cloudbees.com/

```
---
doe: "a deer, a female deer"
ray: "a drop of golden sun"
pi: 3.14159
xmas: true
french-hens: 3
calling-birds:
  - huey
  - dewey
  - louie
  - fred
xmas-fifth-day:
  calling-birds: four
  french-hens: 3
  golden-rings: 5
  partridges:
    count: 1
    location: "a pear tree"
  turtle-doves: two
```

Structured Data Format: JSON

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

- It's based on JavaScript object notation but is language independent. It was standardized in 2013 as ECMA-404.
- *JSON has a much simpler syntax* compared to XML or YAML.
  - Data elements consist of name/value pairs
  - Fields are separated by commas
  - Curly braces hold objects
  - Square brackets hold arrays
- JSON is preferred for communications, data persistence. It is **widely supported** by existing programming languages.

JSON Example

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

Album is a record, containing albums. [ ] denotes an array, { } encloses an object

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>Eng.</dept></employee>
</employees>
```

This is a small record. I had to remove fields to fit the slide.

Serialization

Converting between representations

Serializing Data

We have objects in memory. We’d like to convert them to JSON to save them. How can we accomplish this?

- Serialization is a mechanism to convert a data object to a useful format that you can save/stream or otherwise manipulate outside of your program.
  - **Serialization**: save your object to a stream (file or network).
  - **Deserialization**: instantiate an object from your stream (file or network).

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

var ann = Emp(1001, "Anne Hathaway", "New York")
stream.writeObject(ann) // serialize to a file
```

Reading/Writing Objects to JSON

We can use serialization to convert objects directly into JSON format!

- Serialize data objects into JSON strings.
- Save those strings (aka text) to disk/stream over a network/save to a database.
- Deserialization can be used to reverse the process (convert stream → object in mem)

To add serialization support, install these plugins/dependencies (newest versions):

```
plugins {
   id 'org.jetbrains.kotlin.plugin.serialization' version '1.9.10'
}
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.1"
}
```

```kotlin
@Serializable
data class Project(val name: String, val owner: Account, val group: String)


@Serializable
data class Account(val userName: String)

val moonshot = Project("Moonshot", Account(“Jane”), “R&D”)
val cleanup = Project("Cleanup", Account("Mike"), “Maintenance")

val string = Json.encodeToString(listOf(moonshot, cleanup))
   // [ {“name”:”Moonshot”,”owner":{"userName":"Jane"}, “group":"R&D"},
   //   {“name”:"Cleanup","owner" {“userName":"Mike"},"group":"Maintenance"} ]

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

JSON as a standard

- We'll use JSON for storing and transmitting data anywhere that we need it.
- Structure + data means that we can process it consistently.
  - We can easily convert JSON to/from object format
- Easy to work with it! It's just a string.
  - Read it, print out to the console
  - Save in a text file, using standard File classes
  - Saved to a database
  - Send over a network — *see web services lecture*

Don't underestimate the value in being able to read your data in a debugger, or text editor as you're working with it. JSON being human-readable text is one of its biggest advantages as a data file format.