JSON

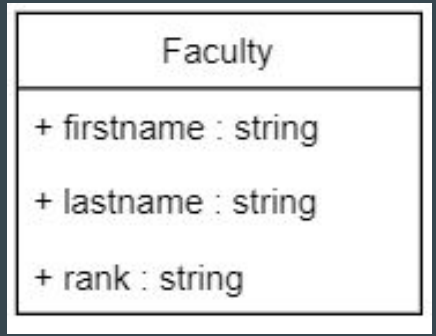
•••

this will get confusing if your name is “jason”

JavaScript Object Notation

JSON is a language for representing data that is an alternative to CSV. It adapts the syntax of JavaScript to define data objects as key-value pairs.

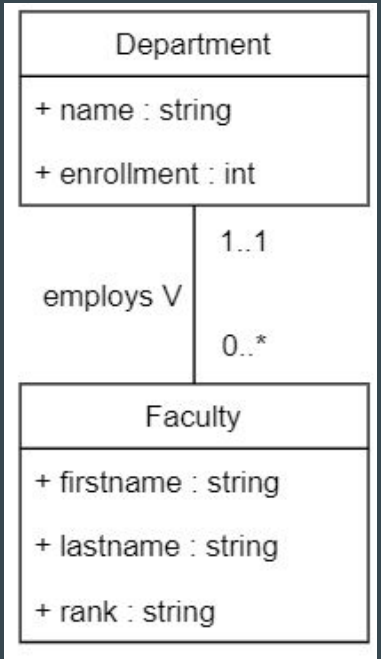
```
{
    "firstName" : "Neal",
    "lastName" : "Terrell",
    "rank" : "Lecturer"
}
```



Keys (attributes) are always strings; values can be integers, floats, strings, Booleans, arrays/lists (with [... , ...] syntax), and objects (key-value pairs, wrapped in curly braces).

Simple Associations via Embedding

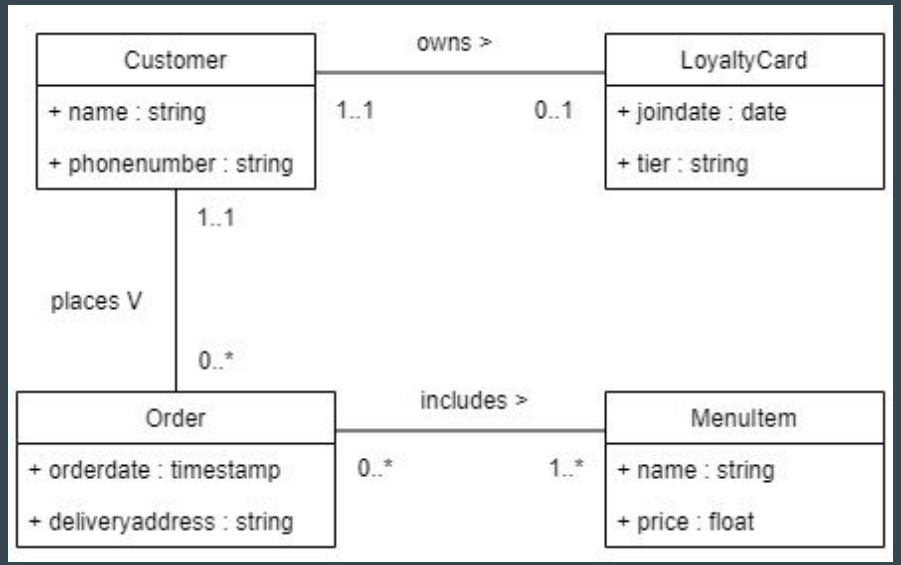
To implement a one-to-one or one-to-many association from UML, we can choose to **embed** the "child" objects directly inside the parent, using a list for a one-to-many:




```
{
    "departmentName" : "Computer Engineering and Computer Science",
    "enrolledStudents": 1253,
    "faculty" : [
        {
            "firstName" : "Neal",
            "lastName" : "Terrell",
            "rank" : "Lecturer"
        },
        {
            "firstName" : "Mehrdad",
            "lastName" : "Aliasgari",
            "rank" : "Associate Professor"
        }
    ]
}
```

```
{
    "name" : "Neal Terrell",
    "phoneNumber" : "562-985-5555",
    "loyaltyCard" : {
        "joinDate" : "2023-01-31",
        "tier" : "Platinum"
    }
}
```

A one-to-one association, directly embedding the “child” object.



Challenge: create a JSON object to represent a car Manufacturer (with a *name* and a *country of origin*), and at least two Models produced by that manufacturer. Models have a *name*, a *model year*, and a *body style* (sedan, compact, minivan, crossover, etc.).

JSON Schema

JSON vs CSV

What's your impression of JSON so far?

One advantage of JSON over CSV is its support for **schemas**: formal descriptions of the structure of the data stored in a file. "JSON Schema" documents are written in JSON itself, and then paired to a specific .json file to describe its contents.

Once a schema is defined, we can use software tools to discover new data sets, validate them, identify errors in the data, and visualize that data – all automatically!

A Simple JSON Schema

Identifies this as a JSON Schema document, not some arbitrary object.

The URL where your schema is published. (We will ignore this.)

Name and describe the data being defined.

This schema defines an object, and not a single primitive value.

All the properties/attributes/fields of the object being defined.

Each property is recursively as another schema, with the same properties as the "root" schema. This defines firstName to be of type string. We could add a description if it would be helpful.

Properties are optional unless they show up in this list.

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://yourdomain.com/schema-url",
    "title": "Faculty",
    "description": "A faculty at the university.",
    "type": "object",
    "properties": {
        "firstName" : {
            "type" : "string"
        },
        "lastName" : {
            "type" : "string"
        },
        "rank" : {
            "type" : "string"
        }
    },
    "required" : ["firstName", "lastName", "rank"]
}
```

Validating against a schema

```
{
    "$schema" : "./faculty.schema.json",
    "firstName" : "Neal",
    "lastName" : "Terrell",
    "rank" : 5
}
```

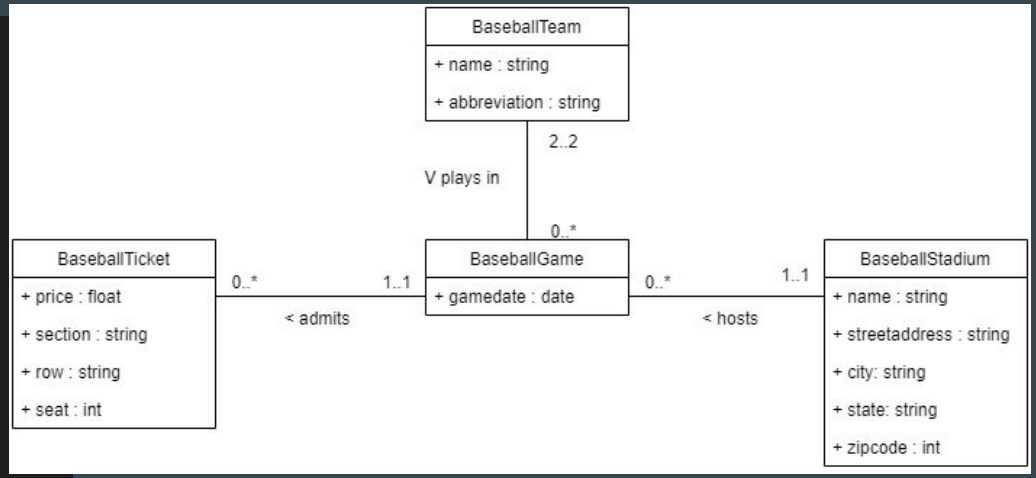
Incorrect type. Expected “string”.

Demo: a complicated schema

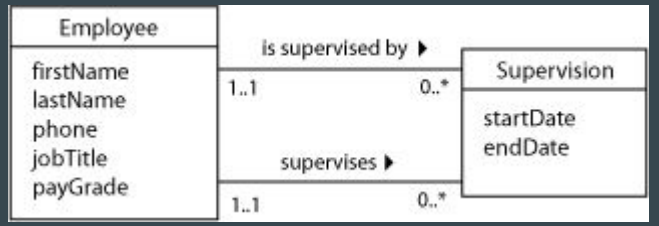
Many-to-many associations

In JSON, an element can only be embedded in a single parent. This is fine for one-to-one/many associations, where the child(ren) have only one parent:

```
{
    "subtotal": 22.49,
    "orderDate": "2022-09-10T16:52:00",
    "pizzas": [
        {
            "size": "medium"
        },
        {
            "size": "large"
        }
    ]
}
```



But what about a class that has many parents, like in a many to many association?



Take the Actor-Film many-to-many, supposing at first we don't also store an actor's *role*. Rule #1: a JSON document must have a single object as the "root", but in a many-to-many, there is no single "parent" class to act as the root.

```
{
    "films": [
        {
            "title": "Top Gun"
        },
        {
            "title": "Mission: Impossible"
        }
    ],
    "actors": [
        {
            "name" : "Tom Cruise"
        },
        {
            "name" : "Ving Rhames"
        }
    ]
}
```

To complete the associations between these objects, we can use either **embedding** or **referencing**.

Any objections???

Many to many, w/ embedding

A many to many between A and B can be solved by embedding the many B under each A object, *and*

```
{
    "films": [
        {
            "title": "Top Gun",
            "releaseYear" : 1986,
            "actors": [
                {
                    "name": "Tom Cruise"
                }
            ]
        },
        {
            "title": "Mission: Impossible",
            "releaseYear" : 1996,
            "actors": [
                {
                    "name": "Tom Cruise"
                },
                {
                    "name": "Ving Rhames"
                }
            ]
```

```
    "actors": [
        {
            "name": "Tom Cruise",
            "films": [
                {
                    "title": "Top Gun",
                    "releaseYear" : 1996,
                },
                {
                    "title": "Mission: Impossible",
                    "releaseYear" : 1996
                }
            ]
        },
        {
            "name": "Ving Rhames",
            "films": [
                {
                    "title": "Mission: Impossible",
                    "releaseYear" : 1996
                }
```

Multiple nesting pros and cons

Pro:

- Easy.
- Can easily find the films for a specific actor, *and also* find the actors for a specific film.

Con:

- Redundancy!!!!!

Many to many, w/ references

If each B (and/or A) has a *key* that uniquely identifies it, we can instead store many *key references* under the A object, and vice-versa.

```
{
    "films": [
        {
            "title": "Top Gun",
            "releaseYear" : 1986,
            "actors": ["Tom Cruise"]
        },
        {
            "title": "Mission: Impossible",
            "releaseYear" : 1996,
            "actors": ["Tom Cruise", "Ving Rhames"]
        }
    ],
```

```
    "actors": [
        {
            "name": "Tom Cruise",
            "films": ["Top Gun", "Mission: Impossible"]
        },
        {
            "name": "Ving Rhames",
            "films": ["Mission: Impossible"]
        }
    ]
}
```

Keys, pros and cons

Pro:

- *Reduced* redundancy. One full copy of each film and actor, and then many *key references*. Ideally keys are short, so duplication is not a big deal!
- Can find all films that an actor is in, and also all of the actors in one film.

Con:

- Still storing both “sides” of the association. (Film stores list of actors; each actor stores list of Films.)
- Eliminating one “side” helps, but makes it harder to find one direction of the association. (Difficult to find all of Tom Cruise’s films.)
- What if your data objects don’t have unique keys? [**Are film titles unique?**]

Many to many, w/ surrogate keys

Sometimes a class doesn't have a simple, small attribute that uniquely identifies each object. We can still use key references if we introduce a **surrogate key** attribute to the class: an *artificial attribute* that we are externally responsible for assigning unique values to.

Integers can work, especially if they are assigned by another authority. *Universally unique identifiers* (UUIDs) are another potential key...


```
{
    "films": [
        {
            "filmId": 1,
            "title": "Top Gun",
            "releaseYear": 1986,
            "actors": [
                "2ac2f3d1-261f-42ad-8cea-5bb370bb8476"
            ]
        },
        {
            "filmId": 2,
            "title": "Mission: Impossible",
            "releaseYear": 1996,
            "actors": [
                "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
                "b281d923-64ea-4e15-8306-6d670875f96d"
            ]
        }
    ],
    "actors": [
        {
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "name": "Tom Cruise",
            "films": [1, 2]
        },
        {
            "actorId": "b281d923-64ea-4e15-8306-6d670875f96d",
            "name": "Ving Rhames",
            "films": [2]
        }
    ]
}
```

Surrogate keys pros and cons

Pro:

- The same as w/ keys, except now every class can have a unique key!
- UUIDs are *universally unique*.

Con:

- The same as w/ keys.
- It can be hard to assign unique IDs to each object, especially in high-volume systems.
- UUIDs are large (16 bytes) and expensive to create.

Many to many, w/ junctions

A **junction** is an object that stores the IDs of the two objects in the relationship. Junctions are stored in another top-level list under the “root” of the document.

Notice how “filmRoles” is playing the role of association class...


```
{
    "films": [
        {
            "filmId": 1,
            "title": "Top Gun",
            "releaseYear": 1986
        },
        {
            "filmId": 2,
            "title": "Mission: Impossible",
            "releaseYear": 1996
        }
    ],
    "actors": [
        {
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "name": "Tom Cruise"
        },
        {
            "actorId": "b281d923-64ea-4e15-8306-6d670875f96d",
            "name": "Ving Rhames"
        }
    ],
    "filmRoles": [
        {
            "filmId": 1,
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476"
        },
        {
            "filmId": 2,
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476"
        },
        {
            "filmId": 2,
            "actorId": "b281d923-64ea-4e15-8306-6d670875f96d"
        }
    ]
```

Junctions pros and cons

Pro:

- Can access both “sides” of the association through the junction attributes.
- The association data is only stored once (instead of twice like w/ keys).
- Minimal redundancy. (*Mathematically* minimal.)

Con:

- Must linear search all junction pairs when searching for one specific object.
    - We’ll improve this later...

Dealing with association attributes

We conveniently ignored the “role” attribute of the Actor-Film relationship. Where do association attributes go in our various many-to-many implementations?

With embedding:

```
{
    "films": [
        {
            "title": "Top Gun",
            "releaseYear": 1986,
            "actors": [
                {
                    "role": "Maverick",
                    "actor": {
                        "name": "Tom Cruise"
                    }
                }
            ]
        }
```

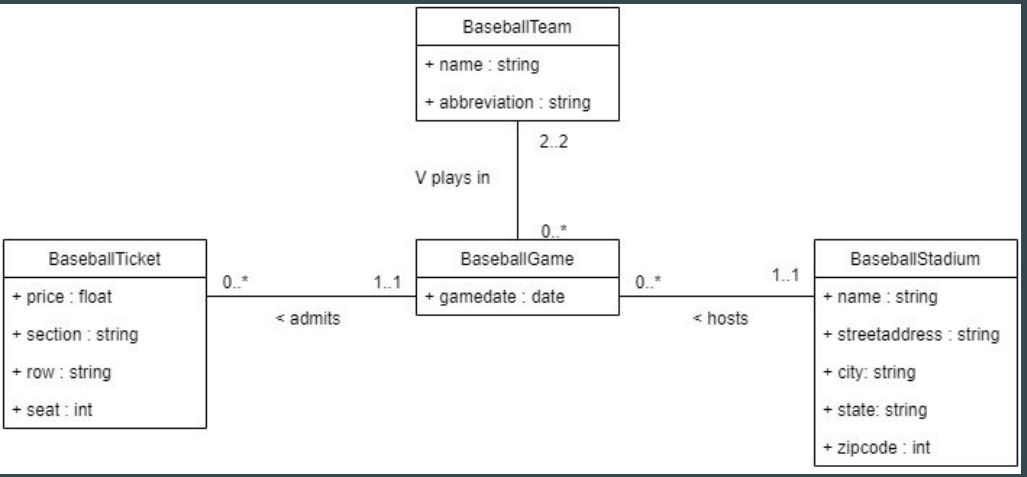
```
"actors": [
        {
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "name": "Tom Cruise",
            "films": [
                {
                    "role": "Maverick",
                    "film": {
                        "title": "Top Gun",
                        "releaseYear": 1986
                    }
                }
            ]
        }
```

With key references:

```
"films": [
    {
        "filmId": 1,
        "title": "Top Gun",
        "releaseYear": 1986,
        "actors": [
            {
                "role": "Maverick",
                "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476"
            }
        ]
    }
],
"actors": [
    {
        "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
        "name": "Tom Cruise",
        "films": [
            {
                "role": "Maverick",
                "filmId": 1
            }
        ]
    },
```

With junctions:


```
{
    "films": [
        {
            "filmId": 1,
            "title": "Top Gun",
            "releaseYear": 1986
        },
        {
            "filmId": 2,
            "title": "Mission: Impossible",
            "releaseYear": 1996
        }
    ],
    "actors": [
        {
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "name": "Tom Cruise"
        }
    ],
    "filmRoles": [
        {
            "filmId": 1,
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "role": "Maverick"
        },
        {
            "filmId": 2,
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "role": "Ethan Hunt"
        },
```




```
{
    "teams": [
        {
            "name": "Los Angeles Angels",
            "abbreviation": "LAA"
        },
        {
            "name": "Seattle Mariners",
            "abbreviation": "SEA"
        }
    ],
    "stadiums": [
        {
            "stadiumid": 1,
            "name": "Angels Stadium",
            "street": "2000 E Gene Autry Way",
            "city": "Anaheim",
            "state": "CA",
            "zipcode": 92806
        }
    ],
    "games": [
        {
            "stadiumId": 1,
            "gameDate": "2017-09-29",
            "homeTeam": "LAA",
            "awayTeam": "SEA",
            "tickets": [
                {
                    "price": 28.0,
                    "section": 530,
                    "row": "C",
                    "seat": 1
                }
            ]
        },
        {
```

JSON’s Place in the World

CSV, revisited

Reconsider the CSV file format that we have seen many times:

```
28.00,Angels,Mariners,Angel Stadium,2017-09-29,C305,H,8
105.00,Angels,Mariners,Angel Stadium,2017-09-29,112,F,1
22.00,Angels,Mariners,Angel Stadium,2017-09-29,C306,FF,18
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,1
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,2
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,3
82.00,Yankees,Angels,Yankee Stadium,2018-07-04,205,EE,4
```

What can we say about these files':

- Structure? [Hierarchical, flattened, indexed, ...?]
- Robustness? [How are they validated? Schemas?]
- Redundancy? [Any way to reduce it?]
- Searchability? [How do you locate a specific data point?]

How about CSV's...

- Accessibility? [Is it human-readable? Do you need special training to use it?]
- Shareability? [How can multiple people share access to the data?]
- Security? [How can we only give access to certain people?]
- Cost? [How much will it cost to address these issues?]

Conclusion: CSV is... bad?

JSON, contrasted

Now let's tear down JSON!

What can we say about these files':

- Structure? [Hierarchical, flattened, indexed, ...?]
- Robustness? [How are they validated? Schemas?]
- Redundancy? [Any way to reduce it?]
- Searchability? [How do you locate a specific data point?]

How about JSON's...

- Accessibility? [Is it human-readable? Do you need special training to use it?]
- Shareability? [How can multiple people share access to the data?]
- Security? [How can we only give access to certain people?]
- Cost? [How much will it cost to address these issues?]

Use cases for CSV and JSON

Where do we actually find these files in the wild?

- CSV:
  - Makes it easy to transmit data between applications.
  - Popular w/ data scientists and business analysts.
  - Microsoft Excel alone is worth the price of using CSV.

- JSON:
  - More work to produce and then consume between applications.
  - But much easier to validate! Less risk of bad data crashing a program.
  - More variety of tools available for viewing and manipulating the data.
  - Often used in configuration files for an application.

What are flat files *bad at?*

- Distributed access.
  - Especially simultaneous read/write.
- Sophisticated manipulation of the data.
  - Creating values, searching for values, aggregating values...
  - Requires an external tool that can't always be integrated into new applications.
- *Partial* access to the data.
  - Must read the entire file to finish a search.
  - Can't grant access to just part of the data.
- Robustness of the data.
  - What happens if you're writing to a file and the power goes out?
- Security.

These are all things that *database managers* excel at, and why we don't simply store all data for all applications in flat files.

Where do we go from here?

We want a *database manager*... a program or library that our applications can interact with, to store, update, and manipulate the data model for our application.

A DBM should excel in all the areas flat files are deficient. We should gain scalable, robust, granular access to **persistent** data, with tools to read and manipulate that data in sophisticated ways. We should be able to define our model using a schema, with automated validation of data to prevent illegal values from ruining our applications. We should be able to purchase a piece of software to do these things for us, and include its cost of license and maintenance in our budget estimates; rather than write the software ourselves.

Easy!