Parsing JSON, Java Collections

Michael Woodley
(with some slides from Carl Evans)
In this course, as well as in the real world, it is not an imperative to make sure that every line of code and every method or function is “optimized for speed”. Most of the time the details of your implementation are not performance-critical because they will not affect the overall performance of your application. Performance is a matter of concern to your decisions in a couple of ways:

1. algorithm
2. implementation

My concern is here is where you should focus your attention: write code that humans can understand.
Separating Data from Code

Imbedding data into your code is rarely good design.

While it IS unrealistic to expect your code to contain NO “data”, let’s clarify things a bit:

- in this discussion, we will consider “data” as information that is acted upon by an application
- size limits, constant values like Pi, etc. are not considered data for this discussion
Separating Data from Code

We will access data that is outside of our code and is not part of the compilation process.

So how will we separate data from our code?

One of the simplest and most common ways to access data from outside our compilation is to read data from a file.
Separating Data from Code

If we are going to read data from a file, should we consider the name of a file to be “data”?

That is, should it be part of compilation?
Separating Data from Code

If we are going to read data from a file, should we consider the name of a file to be “data”?

That is, should it be part of compilation?

Ideally: No
Let’s look at the current assignment and take note of the goals.

https://courses.grainger.illinois.edu/cs126/sp2021/assignments/json/
Parsing JSON in Java

Use the GSON library from Google:

https://github.com/google/gson/blob/master/UserGuide.md

See the section “Using Gson with Maven” here:

https://github.com/google/gson/blob/master/UserGuide.md
#TOC-Gson-With-Maven
Video

Patrick Gallagher’s Excellent JSON Video
Common Data File Formats

CSV, JSON -

are data exchange formats. They are easy to read and construct by humans.

What about XML and HTML?
Common Data File Formats

CSV, JSON -

are data exchange formats. They are easy to read and construct by humans.

What about XML and HTML?

- XML - eXtensible Markup Language is a data exchange format
- HTML is a markup language for use in browsers and not commonly used for data exchange. It is intended for description of representation.
CSV
Comma Separated Values

CSV is perfect for tabular data. It is a plain text format where commas separate values (columns) and lines separate records (rows). Here is an example:

Name, netid, title
Woodley, mwoodley, professor
Elvis, theking1, pet

Great for exchanging tables including spreadsheet data.
JSON

- JavaScript Object Notation
- A lightweight data-interchange format
  - Very commonly used by APIs
- It is easy for
  - humans to read and write.
  - machines to parse and generate.
Example JSON

key: value pairs (remember this)

This is what a single data object looks like within a JSON data file:

{
    "name": "woodley",
    "nickname": "mike",
    "age": 125,
    "title": "professor"
}
Example JSON Data

{

“people”: [ 

{

“name”: “michael woodley”,
“nickname”: “mike”,
“age”: 125,
“title”: “professor”
},
{

“name”: “Elvis”,
“nickname”: “Brindle Boy”,
“age”: 4,
“title”: “dog”
}
]
}
Example JSON Object (Java)

// NOTE: the element names EXACTLY match* the KEYS in the JSON data

class Person {
    private String name;
    private String nickname;
    private int age;
    private String title;

    // methods
}

*this is not necessary but makes your life MUCH easier
Example JSON

More JSON Examples:
https://www.sitepoint.com/10-example-json-files/

Using GSON:
https://www.tutorialspoint.com/gson/gson_quick_guide.htm
Java Collections
(data structures)

- **Collection:**
  - an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**

- **Some types of collections are “ordered” (not sorted):**
  - the concepts of Nth, last and next make sense in these
  - See “Persons Example” spreadsheet
  - some collections maintain an ordering; some allow duplicates
  - typical operations: add, remove, clear, contains (find), size
Java Collections

- Examples of collections that can be found in the Java class libraries:
  - List - ArrayList (ordered)
  - Map - HashMap (NOT ordered)
  - Tree - TreeSet (ordered)

- Collections are in the `java.util` package

```java
import java.util.*;
```
Lists
(Java ArrayList)

- A list is a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based index
  - elements can be added to the front, back, or elsewhere
Java ArrayList and “Generics”

Java Form:
ArrayList<Type> name = new ArrayList<Type>();

Example:
ArrayList<Person> people = new ArrayList<Person>();

When constructing an ArrayList, you must specify the type of elements it will contain between < >.
Java ArrayList and “Generics”

Java Form:

```java
ArrayList<Type> name = new ArrayList<Type>();
```

When constructing an `ArrayList`, you must specify the type of elements it will contain between `< >`.

- The `ArrayList` class can be instantiated to contain a list of any type object.
- This is called a *type parameter* or a *generic* class.
- These types MUST be objects (vs. primitive types)
  - `<int>` is a primitive. Can NOT be used with `ArrayList`
  - use the wrapper class `<Integer>` with `ArrayList`
Boxed Primitive Types

You can not use primitives with ArrayList<int> (or other collections) but Java provides “boxed primitives”:

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
## Some ArrayList Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(value)</code></td>
<td>appends value at end of list</td>
</tr>
<tr>
<td><code>add(index, value)</code></td>
<td>inserts given value just before the given index, shifting subsequent values to the right</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all elements of the list</td>
</tr>
<tr>
<td><code>indexOf(value)</code></td>
<td>returns first index where given value is found in list (-1 if not found)</td>
</tr>
<tr>
<td><code>get(index)</code></td>
<td>returns the value at given index</td>
</tr>
<tr>
<td><code>remove(index)</code></td>
<td>removes/returns value at given index, shifting subsequent values to the left</td>
</tr>
<tr>
<td><code>set(index, value)</code></td>
<td>replaces value at given index with given value</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in list</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>returns a string representation of the list such as &quot;[3, 42, -7, 15]&quot;</td>
</tr>
</tbody>
</table>
Map

Generically known as:
- dictionary
- associative array
- associative container

Maps use:
   Key, Value pairs to associate data

That is:
“keys” are used to access “values”
Map Interface in Java

Map<String, Integer> hm
    = new HashMap<String, Integer>();

hm.put("a", new Integer(100));
hm.put("b", new Integer(200));
hm.put("c", new Integer(300));
hm.put("d", new Integer(400));
Map Interface Methods

- **put(k,v)**  
  Associate v with k

- **get(k)**  
  The value associated with k

- **size()**  
  The number of pairs

- **isEmpty()**  
  Whether it is empty

- **remove(k)**  
  Remove the mapping for k

- **clear()**  
  Remove all mappings

- **containsKey(k)**  
  Whether contains a mapping for k

- **containsValue(v)**  
  Whether contains a mapping to v