Serialization

Jörg Steffen, DFKI

steffen@dfki.de

22.11.2022

equals, == and hashCode

equals vs ==

- Both compare objects in Java
- ==
  - is an operator
  - compares object references
- equals
  - is a method
  - compares object content

```
Student stud1 = new Student("Peter", "Parker");
Student stud2 = new Student("Peter", "Parker");
stud1 == stud2 → false
stud1.equals(stud2) → false
```

equals vs ==

- **`equals`** default implementation in **`Object`**

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

- Overwrite **`equals`**

```
@Override
public boolean equals(Object obj) {
  if (!(obj instanceof Student)) {
    return false;
  }
  Student other = (Student) obj;
  return getFirstName().equals(other.getFirstName())
      && getLastName().equals(other.getLastName());
}
```

HashSet / HashMap and hashCode

- HashSet / HashMap use hashing with collision for efficient access
- Two step approach:
  1. Map (possible infinite) number of objects to a small(er) number of buckets → hashCode
  2. Use linear search in the bucket to see if the element / key exist → equals

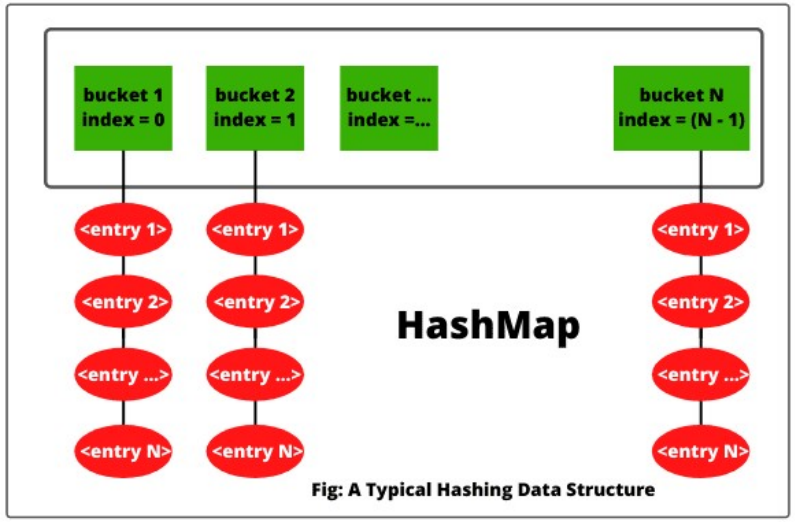

Fig: A Typical Hashing Data Structure

- Object that are equals must have the same hashCode!

hashCode

- Overwrite **`hashCode`**

```
@Override
public int hashCode() {
   return Objects.hash(this.firstName, this.lastName);
}
```

Serialization

What is Serialization?

- Process of translating an object into a format that can be stored or transmitted
  - file
  - XML
  - JSON
  - …
- Serialized object can be reconstructed

  → Deserialization
- Serialized objects do not contain their methods and static fields

Java Build-in Serialization

- Serialization as sequence of bytes
- Classes require marker interface **`Serializable`**
  - Referenced classes must implement **`Serializable`** too
- Exclude fields with **`transient`** keyword
- Special field **`long serialVersionUID`**
  - Unique class identifier
  - Ensure compatibility when deserializing class
  - Automatically created by JVM but better set yourself

```
public class Student implements Serializable {
  private static final long serialVersionUID = 168613573991523828 9L;
  private String firstName;
  private String lastName;
  private transient int age;
  …
```

Java Build-in Serialization

- Use **`ObjectOutputStream`** for writing objects

```
public final void writeObject(Object obj)
```

- Use **`ObjectInputStream`** for reading objects

```
public final Object readObject()
```

→ must be cast back to original type

```
Student stud = (Student) in.readObject();
```

- You can write multiple objects to the same output stream
  - ➢ just make sure you read them back in the same order

Java Build-in Serialization

- Disadvantages
  - Java only
  - binary format
  - fragile when class is edited

JSON

- JavaScript Object Notation
  - widely used interchange format
  - derived from JavaScript
  - human readable
  - language independent
- Syntax
  - attribute-value pairs
  - attributes are string
  - values are
    - Number
    - String
    - Boolean
    - Array (also on top-level): in square brackets, comma separated
    - Object (collection of attribute-value pairs): in curly brackets, comma separated

JSON

```
{
  "name": "Donald Duck",
  "isReal": false,
  "age": 30,
  "address": {
    "city": "Duckburg",
    "country": "USA"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "nephews": [ "Huey", "Dewey", "Louie" ]
}
```

JSON Serialization Using Jackson

- Add **`jackson-databind`** as dependency

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.14.0</version>
</dependency>
```

- Use **`ObjectMapper`** for writing and reading objects

```
ObjectMapper objectMapper = new ObjectMapper();
```

- Use annotations to configure Jackson serialization
  - `@JsonIgnore` → Ignore single field
  - `@JsonInclude(Include.NON_NULL)` → Ignore null value fields

JSON Serialization Using Jackson

- Writing Java object as JSON

```
objectMapper.writeValue(new File("stud.json"), stud);
String json = objectMapper.writeValueAsString(stud);
```

→ {"firstName": "Peter", "lastName": "Parker"}

- Reading Java objects from JSON

```
Student stud = objectMapper.readValue(
    new File("stud.json"), Student.class);
Student stud = objectMapper.readValue(json, Student.class);
```

JSON Serialization Using Jackson

- For collections, use **`TypeReference`** to specify type

```
List<Student> studList =
    objectMapper.readValue(
        jsonStudArray, new TypeReference<List<Student>>(){});
```

- Every non-array JSON string can be parsed into a Java Map

```
Map<String, Object> map =
    objectMapper.readValue(
        json, new TypeReference<Map<String, Object>>(){});
```