# Java Object-Oriented Concepts Unit

## Lesson 6: Data Encoding and Decoding

The Learning House

427 S 4th Street #300

Louisville KY 40202

# Lesson 6: Data Encoding and Decoding

## Overview

Up to this point, we have been dealing with data as variables (primitives and objects) in memory and occasionally reading values from the console (and then promptly storing these values in a variable). This has worked well, but all of our data disappears when the program completes. In order to write truly useful programs, we must be able to store data when program execution ends and read that data in next time we run our programs.

In this lesson, we will begin to explore data persistence and ways that we can encode data from variables (in memory) to files (on disk) for long-term storage and then reverse the process when we need to access the data later. We will concentrate on storing data in flat text files for now, but the principles of this lesson continue to be relevant when we start storing data in relational databases.

## Data Storage and Representation

We have two options available to us for data storage: we can store data in memory, which is volatile, or we can store the data on more permanent media such as a hard disk, flash drive, or optical drive. Java (and other languages) has libraries that allow us to write data to and read data from more permanent media. This is convenient for us, but these libraries don't have much intelligence — they will simply write out the bits that we give them. It is up to us to decide how this data should be represented and encoded.

A group of related data is generally represented by an object in object-oriented languages such as Java. The object itself represents the thing being modeled (e.g. a Student) and the fields on the object represent particular properties of the thing being modeled (e.g. first name, last name, age). In past lessons, we have seen how to create these classes and instantiate them into objects. We have also seen how to set values for the properties on the objects and even how to store them in Maps and Lists. We have not had to do any translation or encoding/decoding of any of these values because everything has always been stored as an object in memory. Now that we wish to store this data, we must decide how the data will be represented. As mentioned in the overview, for now, we will be storing our data in flat text files on our hard drives. We must encode the objects we have in memory in some way so that we can store them.

Our choice of using flat text files as our storage format implies that all of our data (even numbers) will be stored as text. We must design a file format that allows us to easily read from and write to the file and that allows us to easily tell where the data of one object ends and the next one begins. We must also keep track of which fields are text and which fields are numbers so that we can convert them properly when reading from the file.

## File Format, Encoding, and Decoding

To illustrate file format, encoding, and decoding, we will use the example of storing a student class roster to a file. We will discuss the particulars of the file format here and leave the details of the implementation to the live coding session in class. A complete solution (with annotated source code) will also be posted to the class GitHub repository.

The key characteristics of any file format we choose to implement are:

- We must be able to easily tell where one student record ends and the next begins.
- We must be able to easily tell where one property within a student record ends and the next begins.

- Malformed records should have little or no impact on our ability to properly read subsequent records.
- The format must make it straightforward to read from and write to the file; in other words, the format should be easy to parse.

Given these overall requirements, we will go with the following file format:

1. Each line in the file represents one student — this satisfies #1, #3, and #4 above
2. Each field in the student record will be separated with the token "::" — this satisfies #2 above.

As a bonus, we can tell how many students are in the roster by the number of lines in the file.

## Student Class

The Student class shows how student data is stored in memory:

```
public class Student {

    private String firstName;
    private String lastName;
    private String studentId;
    private String cohort;   // Java or .NET + cohort month/year

    public Student(String studentIdIn) {
        studentId = studentIdIn;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getStudentId() {
        return studentId;
    }
    public String getCohort() {
        return cohort;
    }
    public void setCohort(String cohort) {
        this.cohort = cohort;
    }
}
```

## File Format Example

Our file format illustrates how student data is stored in the file:

```
<student id>::<first name>::<last name>::<cohort>
```

For example:

```
0001::John::Doe::Java - August 2014
0002:: Sally::Smith::Java - April 2014
0003:: John::Jones::.NET - Jan 2014
```

## Encoding and Decoding Approach

Now that we know how our student data will be represented in memory and on the disk, we can write code that translates from one to the other. Here is the outline for writing student data to the file:

1. Open the file for writing
2. Go through the collection of students one by one
3. For each student, do the following:
    a. Create a String consisting of student id, first name, last name, and cohort (in that order), separated by ::
    b. Write the String to the output file
    c. Get the next student (if exists) and go back to step a
    d. If there are no more students to process, quit
4. Close the file

Here is the outline for reading student data from the file:

1. Open the file for reading
2. For each line in the file, do the following:
    a. Read the line into a String variable
    b. Split the String into chunks at the :: delimiter
    c. Create a new Student object
    d. Use the first value from the split string to set the student id
    e. Use the second value from the split string to set the student first name
    f. Use the third value from the split string to set the student last name
    g. User the fourth value from the split string to set the cohort value
    h. Put the newly created and initialized Student object into a Collection or Map
    i. If there are more lines in the file, go to step a
    j. If there are no more lines in the file, quit creating student objects
3. Close the file

This high-level encoding/decoding approach will continue to be useful even as we move from storing our data in flat file to storing data in relational databases.  We will use different tools and libraries, but the general concepts will remain the same.