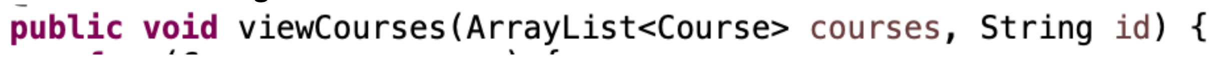
Overall Design of CRS

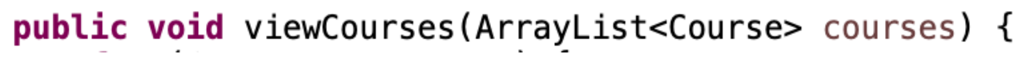
Tianhao Gu

The main method is contained in the CRS Class. When it runs, the program first determines if it is the first time of logging in by checking the existence of the serialization file. If so, the program will read from the csv file and fill in the empty course list. If no, the program will operate deserialization and read the data object, then points the student list and the course list to the variables the data object contains. Then the program will ask the user to choose an identity (or exit, no serialization will be operated at this stage). A user must enter the username and password both correctly to log in, or the program will prompt the user to enter again or return to the identity selection in case the user selected the wrong option. After login, the corresponding user's login status will be set to 1 in the super class. Then, the menu for either type of user will be output by checking the actual type of the user again. The user can input number to do operations they want following the menu. After an operation is finished, the program will output the previous menu again. The user can do another operation or selects to return to the main menu. At logout or exit, the logins status will be reset to 0.
The overall structure of the CRS is shown in the UML Diagram.
The following is the example of conception use:

Method overloading:



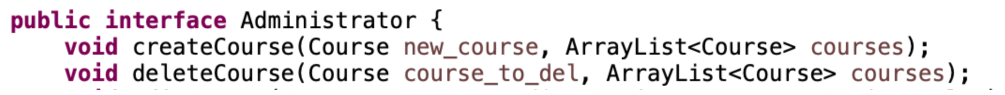
```java
public void viewCourses(ArrayList<Course> courses, String id) {
```



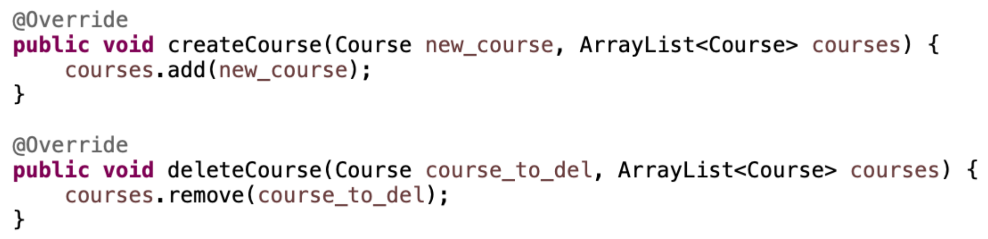
```java
public void viewCourses(ArrayList<Course> courses) {
```

The two viewCourses () methods in Admin class are overloading since they have the same name and signature but take different parameters. The one above is used to display info for a single class so that it takes an extra parameter id. The one below is used to show all info.

Method overriding:



```java
public interface Administrator {
    void createCourse(Course new_course, ArrayList<Course> courses);
    void deleteCourse(Course course_to_del, ArrayList<Course> courses);
```

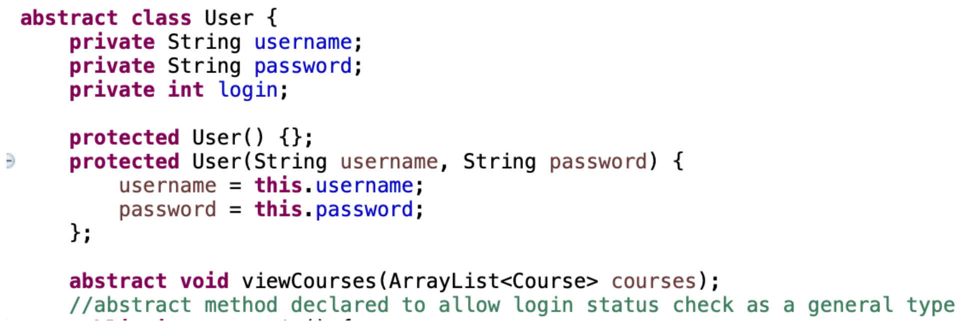


```java
@Override
public void createCourse(Course new_course, ArrayList<Course> courses) {
    courses.add(new_course);
}

@Override
public void deleteCourse(Course course_to_del, ArrayList<Course> courses) {
    courses.remove(course_to_del);
}
```

The Admin class implements the Administrator Interface. It has to implement all the methods in the interface. The createCourse():void method and deleteCourse(): void in the Administrator Interface are abstract. The methods in the Admin class will be called if the Admin selects to create or delete a course. The two methods are overriding.

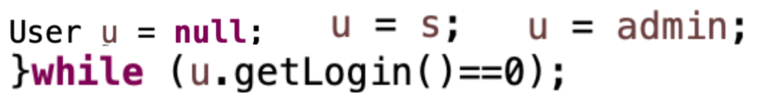
Abstract Class & ADT:

In my design, the user class is set abstract since all practical instantiation is done in the actual type. The user class contains one abstract method and two concrete methods except for the constructors.



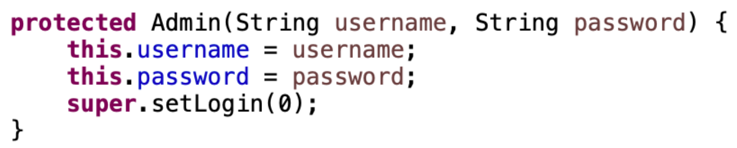
```
abstract class User {
    private String username;
    private String password;
    private int login;

    protected User() {};
    protected User(String username, String password) {
        username = this.username;
        password = this.password;
    };

    abstract void viewCourses(ArrayList<Course> courses);
    //abstract method declared to allow login status check as a general type
```

I separate the login process and the operations. Using abstract type, the program manages to check the login status of the current user (s for student, or admin) as a general type to realize the function of logout.



```
User u = null;    u = s;    u = admin;
}while (u.getLogin()==0);
```

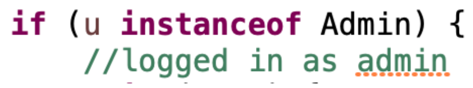
Inheritance:



```
protected Admin(String username, String password) {
    this.username = username;
    this.password = password;
    super.setLogin(0);
}
```

In the constructors of Admin and Student, an object is default to have login status 0. Since both share same login logic, I put the login variable in the User class. The login status is set and accessed from the parent class using super.

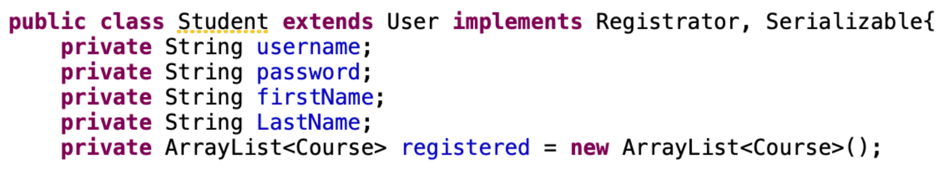
Polymorphism:



```
if (u instanceof Admin) {
    //logged in as admin
```

I use the abstract User data type to track the login status of a user. After the login, the actual type of the logged-in User will be checked again with instanceof and the corresponding menu will be displayed.

Encapsulation:

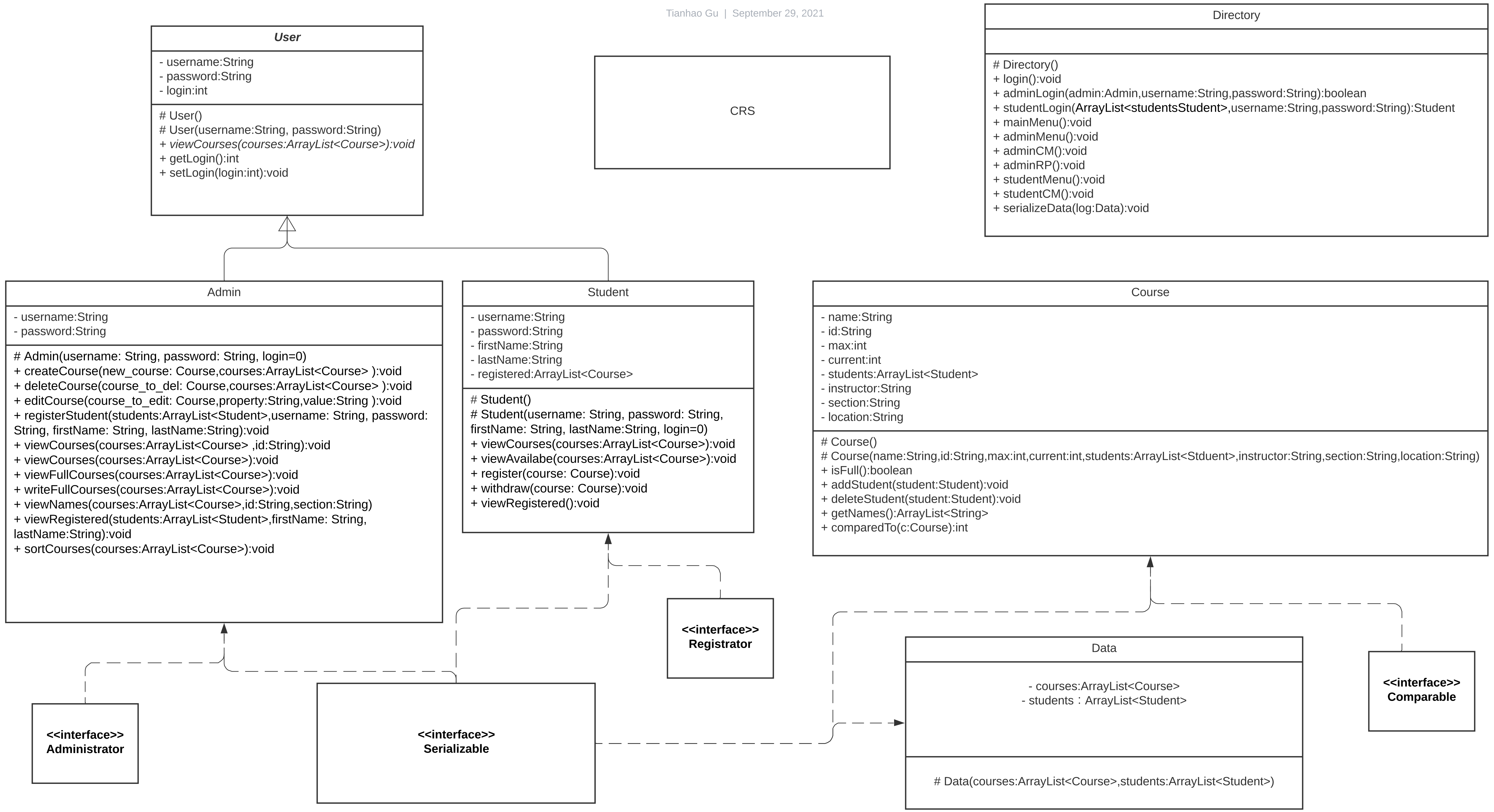


```
public class Student extends User implements Registrator, Serializable{
    private String username;
    private String password;
    private String firstName;
    private String LastName;
    private ArrayList<Course> registered = new ArrayList<Course>();
```

The class variables in my design are all set private. They can only be accessed with getters outside the class. It helps keep personal information and school data safe.

UML Diagram of the CRS

Tianhao Gu | September 29, 2021



```
User
- username:String
- password:String
- login:int
# User()
# User(username:String, password:String)
+ viewCourses(courses:ArrayList<Course>):void
+ getLogin():int
+ setLogin(login:int):void
```

```
CRS
```

```
Directory

# Directory()
+ login():void
+ adminLogin(admin:Admin,username:String,password:String):boolean
+ studentLogin(ArrayList<studentsStudent>,username:String,password:String):Student
+ mainMenu():void
+ adminMenu():void
+ adminCM():void
+ adminRP():void
+ studentMenu():void
+ studentCM():void
+ serializeData(log:Data):void
```

```
Admin
- username:String
- password:String
# Admin(username: String, password: String, login=0)
+ createCourse(new_course: Course,courses:ArrayList<Course> ):void
+ deleteCourse(course_to_del: Course,courses:ArrayList<Course> ):void
+ editCourse(course_to_edit: Course,property:String,value:String ):void
+ registerStudent(students:ArrayList<Student>,username: String, password:
String, firstName: String, lastName:String):void
+ viewCourses(courses:ArrayList<Course> ,id:String):void
+ viewCourses(courses:ArrayList<Course>):void
+ viewFullCourses(courses:ArrayList<Course>):void
+ writeFullCourses(courses:ArrayList<Course>):void
+ viewNames(courses:ArrayList<Course>,id:String,section:String)
+ viewRegistered(students:ArrayList<Student>,firstName: String,
lastName:String):void
+ sortCourses(courses:ArrayList<Course>):void
```

```
Student
- username:String
- password:String
- firstName:String
- lastName:String
- registered:ArrayList<Course>
# Student()
# Student(username: String, password: String,
firstName: String, lastName:String, login=0)
+ viewCourses(courses:ArrayList<Course>):void
+ viewAvailabe(courses:ArrayList<Course>):void
+ register(course: Course):void
+ withdraw(course: Course):void
+ viewRegistered():void
```

```
Course
- name:String
- id:String
- max:int
- current:int
- students:ArrayList<Student>
- instructor:String
- section:String
- location:String
# Course()
# Course(name:String,id:String,max:int,current:int,students:ArrayList<Stduent>,instructor:String,section:String,location:String)
+ isFull():boolean
+ addStudent(student:Student):void
+ deleteStudent(student:Student):void
+ getNames():ArrayList<String>
+ comparedTo(c:Course):int
```

```
Data
- courses:ArrayList<Course>
- students：ArrayList<Student>
# Data(courses:ArrayList<Course>,students:ArrayList<Student>)
```

<<interface>> Registrator
<<interface>> Administrator
<<interface>> Serializable
<<interface>> Comparable