Java Communication: Chapter 5

Classes, Objects, Constructor, Instance variables and methods, Overloading

Contents:

Classes

Blueprints for objects. Objects are constructed out of classes.

- Holds methods and variables exclusive to those objects (encapsulation)
- Can have parameters for how an object is constructed (construction methods)
- Can have instances of objects
    - Each object can have different variables (instance variables)
- Sometimes classes can have their own unique folder if they are public static

```
class Athlete {
private double weight;
private double height;
private int age;
private double[] calories;
private String firstName;
private String lastName;

// Constructing athlete class

    public Athlete(String firstName, String lastName, double weight, double height, int age) {
        this.weight = weight;
        this.height = height;
        this.age = age;
        this.calories = new double[7];
        this.firstName = firstName;
        this.lastName = lastName;
    }

// Method to store inputted data within an athlete object

    public void inCal (int day, double calBurn) {
        if (day >= 0 && day <=7) {
            calories[day]=calBurn;
        }
    }

// Method to store calories burned per day within athlete object

    public int getAge (int age) {
        return age;
    }

// Method to store get age

    public double avgCal () {
        double sum = 0;
        double avg = 0;
        for (int runs = 0; runs < calories.length; runs++) {
        sum += calories[runs];
        }
        avg = sum/calories.length;
        double avground = Math.round(avg*Math.pow(10,2))/Math.pow(10,2);
        System.out.println("Athlete's average calories burned: " + avground);
```

This is the athlete class. To be created, it requires a constructor, "public Athlete." Class-specific methods that work with athlete objects, like "getAge," return the value attached to a private variable within the class "inCal," which creates an array to be attached to each athlete.

Objects + Constructors

Object:

Objects are constructed out of classes (blueprints). Think of objects as iterations of a class. (Instantiation)

- Each object can hold its own data field (variables).
    - The variables are called instance variables
- Can be accessed by methods within their own class

Constructor:

- A method that instantiates a new instance of an object
- Parameters within a constructor method can attach values to the data field within an object
    - Instances of variables within an object
- The constructor initializes the object, which creates space on the heap
- Constructors must always be named the same as the class
- The keyword "new" accesses the constructor effectively constructing a new object

```
class Gym {

private String name;
private int numAthletes;
private Athlete[] athletes;

    public Gym (String name, int maxAthletes) {
        this.name = name;
        this.numAthletes = 0;
        this.athletes = new Athlete[maxAthletes];
    }
```

"public Gym" is the constructor method; notice it has the same name and naming conventions as the "Gym" class itself. It has the parameters of (name, maxAthletes) and gives the object being constructed a new instance of three variables: "name," "numAthletes," and "athletes" (signified by the "this." keyword)

```
Gym gym = new Gym("Elite Fitness", athleteQuant);
```

This is the object being constructed (indicated by the "new" keyword). The "Gym" object is being constructed with the name "Elite Fitness" and a max of "athleteQuant," which refers to an array object. This means the "maxAthletes" value is the size of an array of athletes.

Instance Variables, Instance Methods, and Overloading

Instance Variables:

- Denoted by "this." Keyword
- Every object has these variables, but they likely have different values within each field
- Could be non-primitive or primitive
- Only exist in memory with objects
    - Objects must have been initialized for instances of variables to be allocated in memory
- Most defined within constructor

- Operates within heap (attached to objects)

```
public Gym (String name, int maxAthletes) {
    this.name = name;
    this.numAthletes = 0;
    this.athletes = new Athlete[maxAthletes];
}
```

These are instance variables within a constructor. Indicated by "this.," these variables have an initial value of null (no value), and they will only have a value allocated in memory when an object is created with a new instance.

Instance Methods:

- Exact same characteristics as instance variables
- Can access instance variables
- Operates with objects' instances
- Operates within heap (attached to object

```
public void saveReportToFile(PrintWriter output) {
    for (int runs = 0; runs < numAthletes; runs++) {
        athletes[runs].athleteSummary(runs);
      output.println("Summary of Athlete " + (runs + 1));
      output.println("Name: " + athletes[runs].getFirstName() + " " + athletes[runs].getLastName());
      output.println("BMI: " + athletes[runs].getBMI());
      output.println("Max Heart Rate: " + athletes[runs].getMaxHeart((athletes[runs]).getAge()));
      output.println("Average Calories: " + athletes[runs].avgCal());
       }
    }
```

This is an instance method. It will print out a summary of calculations utilizing instance variables, or instance variables themselves, all of which are attached to an object. This method works on objects themselves, not the class, and uses "." to be called upon.

Overloading:

Overloading refers to overloading a method. Multiple methods will exist under the same name but have different method signatures.

- Different return types or different parameters

```java
public static int positiveInput (int validinput) {

    Scanner valuein = new Scanner(System.in);
    int value = 0;

    do {

    if (!valuein.hasNextInt()) {
    System.out.println("Invalid entry, enter a whole number");
    valuein.nextInt(); }
    value = valuein.nextInt();
        if (value < 0) {
            System.out.println("Invalid entry, enter a positive number"); }
    }
    while (value < 0);
    System.out.println("Entry accepted");

    return value;

    }

// Verifies positive int

public static double positiveInput (double validinput) {

    Scanner valuein = new Scanner(System.in);
    double value = 0;

    do {
        if (!valuein.hasNextDouble()) {
    System.out.println("Invalid entry, enter a whole number");
    valuein.nextDouble(); }
    value = valuein.nextDouble();
        if (value < 0) {
            System.out.println("Invalid entry, enter a positive number"); }
        }
    while (value < 0);
    System.out.println("Entry accepted");

return value;
    }

// Verifies positive double
```

Here are two "positiveInput" methods that are both public and static. They receive the "validinput" value within their method signature. They are methods that check to make sure the input value is positive. However, they have different return types and data types; one possesses a data type of double within its signature, the other possesses an integer data type within its method signature. One verifies an integer, the other verifies a double.

Object instantiation and accessing object's data fields and their methods

Object Instantiation:

- A new instance of an object is created
    - Like a new instance of a variable
- This is denoted by the "new" keyword
- This new instance of an object within a class may have new instances of variables

Access object data fields and methods:

- Object data fields can sometimes be private within its class (encapsulated)
- To access private data fields getters and setters must be used
- Private data fields can be turned into instance variables if declared within their respective class – This makes it so that the initial data fields cannot be edited, but new instances can be edited
    - Constructor method can edit these values

```
class Athlete {
private double weight;
private double height;
private int age;
private double[] calories;
private String firstName;
private String lastName;
```

Initial private encapsulated variables are created unique to the athlete class (private).

```
public Athlete(String firstName, String lastName, double weight, double height, int age) {
    this.weight = weight;
    this.height = height;
    this.age = age;
    this.calories = new double[7];
    this.firstName = firstName;
    this.lastName = lastName;
}
```

The constructor method creates new instances of the private variables within the new athlete object (this.)

```
athletes[runs] = new Athlete( firstName, lastName, weight, height, age);
```

A new athlete object is constructed at index [runs], and with its creation, new instances of the variables are also created. This is due to the Athlete constructor method having those variables within its perimeter.

```
System.out.println("Enter athlete first name ");
String firstName = valuein.next();

System.out.println("Enter athlete last name ");
String lastName = valuein.next();

// Input first and last name

System.out.println("Enter weight for athlete " + (runs + 1) + " (in lbs)");
double weight = positiveInput(valuein.nextDouble());

System.out.println("Enter height for athlete " + (runs + 1) + "(in inches)");
double height = positiveInput(valuein.nextDouble());

System.out.println("Enter age for athlete " + (runs + 1) + "");
int age = positiveInput(valuein.nextInt());
```

The user inputs the values of the new instance variables.

Encapsulation, getters, and setters

Encapsulation:

Encapsulated data fields (variables), encapsulated methods, etc. are all enclosed within a class. These traits can only be accessed within the class they are enclosed in, unless in some cases, getters or setters are used.

Private variables and methods:

- The "private" keyword encapsulates variables and methods
  - Encapsulated variables and methods
    - Can only be accessed within the class
    - Getters and setters can be used to access a modified version of the private variable
    - Encapsulated methods cannot be accessed outside of class

```
class Athlete {
private double weight;
private double height;
private int age;
private double[] calories;
private String firstName;
private String lastName;
```

These private variables are variables encapsulated within the Athlete class. They can only be accessed either in the class or outside of the class via getters and setters like "setAge" or "getAge."

Getters:

- Method to return a value
  - Returns a value within a private variable
  - Allows access to private variables outside of class
  - Getters must be defined within the class of private variables

```
athletes[runs].getMaxHeart((athletes[runs]).getAge()));
```

The "getMaxHeart" method requires "age," so to access the private variable age of the object at Athletes[runs], the "getAge" method must be used to return the value attached to age. (The "." In ".getAge()" refers to accessing a method and using it on an object.

```
public int getAge (int age) {
    return age;
}
```

The "getAge" method returns the age, and since age is an integer, the return type is int.

Setters:

- Method to edit a value
  - Edits a value within a private variable
  - Allows access to private variables outside of class
  - Setters must be defined within the class of private variables

```
public void setAge () {
if (age > 0)
    this.age = age;
    else {
        System.out.println("Invalid age value");
    }
}
```

The set age method checks to make sure that the age value is greater than zero. If it is, the private variable age's new value will be set. If not, it will not be set, and the console will print "Invalid age value" instead.

```
athletes[runs].setAge();
```

This utilizes the set age method to change the private variable age within main.

Object reference variables + memory on stack and heap

Reference variables:

- Reference variables are addresses
    - Addresses that direct to a certain spot on the heap
        - Reference variables are stored on stack
        - "Spot on heap" refers to a certain object on the heap

```
Gym gym = new Gym("Elite Fitness", athleteQuant);
```

"Gym gym" is a reference variable. "new Gym("Elite Fitness," athleteQuant)" shows an object being constructed with new instances. "Gym gym" directs to the spot on the heap where "new Gym" resides.

Stack + Heap:

- Heap
    - All objects are stored on the heap
        - Arrays, new gyms, new athletes
            - Initial space is allocated for array size
    - All instance variables are stored on the heap
        - New instances are created with new objects
- Stack
    - Methods, reference variables, primitive variables are stored on the stack
        - Once the code has run, the stack will then collapse
            - Space is not allocated until methods or local variables are initialized
                - Uninitialized variables are null valued

Array of objects

Object Arrays:

- Arrays can hold objects
    - Promotes DRY – reduces repeatedly initializing objects
    - Each object can hold instances of variables
    - Can be used within a loop
    - Drastically improves code's readability

```
athletes[runs] = new Athlete( firstName, lastName, weight, height, age);
```

This method updates the athlete object at x index with a new instance and a set of new instances of variables. Additionally, it saves data to each object, and each index has its own instance of variables

```
public Athlete(String firstName, String lastName, double weight, double height, int age) {
    this.weight = weight;
    this.height = height;
    this.age = age;
    this.calories = new double[7];
    this.firstName = firstName;
    this.lastName = lastName;
}
```

This is the constructor method that will create a new instance of an object. "this." refers to "this instance"

```
Athlete[] athletes = new Athlete [athleteQuant];
```

Here is the creation of the array with athlete objects. "Athlete[] athletes" establishes a reference variable. "Athlete" is a reference to a certain spot in the Athlete array.

```
for (int day = 0; day < 7; day++) {
    double calBurn = positiveInput();
    athletes[runs].inCal(day, calBurn);

}
```

This is an example of an array of objects being used within a loop. The array of calories has seven indexes, so the "for" loop will run seven times. For each of the seven days, the variable "calBurn" will be updated with a new input. Thus, the calories array within each athlete at index x will be effectively updated