Michael Mills

July 3, 2023

Mid-Semester Project

Chapter 1: Getting Started with Java

Java is a very important language to learn, especially in the context of learning object oriented programming. One reason for why Java is an important language to learn, especially in the context of learning object oriented programming is due to the fact that Java is widely used across the industry. In other words, there are many companies throughout the world that have through products and services built largely in Java. Another reason is due to the fact that Java is entirely an object oriented programming language. Alternatively, there are many languages such as C++ that can be used either as an object oriented programming language or a functional programming language. By teaching object oriented programming in a language that is based 100% on an object oriented programming philosophy, it forces students to not stray from the path and remain in an object oriented mindset when building applications.

There are certainly a number of differences between Java and Python. One of which is the way that both of the languages are compiled. As for Java, we first begin with a .java file, which is a file that is readable to humans. We then have a compiler that will take the content of the .java file and convert it into bytecode, which is represented in .class files. Next, the bytecode is loaded and is compiled even further into code that an operating system can use. This will vary depending on what type of operating system is being used. In contrast, Python is a much different programming language in terms of the compilation process. To begin, we start off with a .py file which contains human-readable code. We then have something called an interpreter which will take the content of the .py file and compile it into byte code. The byte code will oftentimes be stored under __pycache__ files. Next, the python interpreter will then read the byte code and perform whatever tasks that the code describes. With that being said, Python skips the process of breaking the bytedown even further into machine language, and keeps it in byte code.

Although there are a number of differences between Java and Python, we can also draw a lot of similarities in the way in which it looks syntactically. For example, we can create a class called comments_and_syntax. This block of code simply prints out the text “Let’s play tennis!” I have also included two different types of comments in the code as well. I have included both the multi line comment as well as a single line comment.

```
// This is an example of a one line comment
/*
    This is an example
    of
    a multi line comment
```

```java
*/

// Class definition
public class Comments_and_Syntax {

    // Main method to run the program
    public static void main(String[] args) {
        System.out.println("Let's play tennis!");
    }
}
```

Comments_and_Syntax.java

There are certainly a number of data types that are included in Java. Some of the data types that are included in Java are int, float, Boolean, and char. These different data types all represent a specific amount of predefined memory. One thing that is important to note is that because Python can be defined as an interpreted language, or a language that uses an interpreter to run the code, there are some differences in the way in which we can define types. Since Python uses an interpreter, it allows for something called dynamic programming, which removes the necessity of explicit type declarations. In contrast, Java does not have this functionality.

For example, we can create a class called data_types that will essentially define all of the main data types within Java. In this particular example of code, we define all of the different data types, and then we print them out.

```java
public class data_types {
    public static void main(String[] args) {
        int player1Score = 3;
        int player2Score = 2;
        float matchDuration = 1.5f;
        boolean isTiebreak = false;
        char winnerInitial = 'A';

        System.out.println("Tennis Match Result:");
        System.out.println("Player 1 Score: " + player1Score);
        System.out.println("Player 2 Score: " + player2Score);
        System.out.println("Match Duration: " + matchDuration + " hours");
        System.out.println("Is Tiebreak? " + isTiebreak);
        System.out.println("Winner Initial: " + winnerInitial);
    }
}
```

Data_types.java

Another important concept that is specifically related to data types is a concept called type casting. Type casting can be defined as the process of taking one data type and converting the data type into another type. This is primarily helpful when a particular data type that we are using does not have the functionality that we need.

For example, we can create a class called type_casting that will take a certain data type and change it to a different data type. In this particular instance, we are able to change from an int to a float, and then from a float to an int.

```java
public class type_casting {
    public static void main(String[] args) {
        int playerScore = 6;
        float averageScore = 8.5f;

        // Type casting from int to float
        float convertedScore = (float) playerScore;

        // Type casting from float to int
        int roundedAverage = (int) averageScore;

        System.out.println("Player Score: " + playerScore);
        System.out.println("Converted Score (float): " + convertedScore);
        System.out.println("Average Score (float): " + averageScore);
        System.out.println("Rounded Average Score (int): " + roundedAverage);
    }
}
```

Type_casting.java

Chapter 2: Object Oriented Design with Class Objects

Having a solid understanding of classes and how they operate is essential to becoming a successful Java developer. In java and in object oriented programming, a class can be defined as a template for various objects that we are looking to create. A class will define all of the different attributes and behaviors that an object may have.

Constructors are another important concept in the realm of object oriented programming. In Java, a constructor can be defined as a way in which we can initially set up an object when it is first created. In other words, when an object is first created, we are able to define all of its attributes through a constructor.

For example, let's say for instance that we have a class called constructor, that contains a constructor. The constructor will take in two parameters, name and age. We then have the getName() and getAge() methods that will then be able to get the values of those particular attributes.

```
public class constructor {
    private String name;
    private int age;

    public TennisPlayer(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public static void main(String[] args) {
        TennisPlayer player = new TennisPlayer("Roger Federer", 39);
        System.out.println("Name: " + player.getName());
        System.out.println("Age: " + player.getAge());
    }
}
```

Constructor.java

Getters and setters are another important concept in the relam of object oriented programming. In Java, getters and setters can be defined as a way in which we are able to modify attributes that are assigned to a particular class.

For example, let’s say for instance that we have a class called getter_and_setter that has a number of private attributes, age and name. In this class, we have getter and setter methods that return the corresponding attributes.

```
public class getter_and_setter {

    private String name;
    private int age;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public static void main(String[] args) {
        TennisPlayer player = new TennisPlayer();
        player.setName("Roger Federer");
        player.setAge(39);
        System.out.println("Name: " + player.getName());
        System.out.println("Age: " + player.getAge());
    }
}
```

Getter_and_setter.java

In conclusion, object oriented design, classes, constructors, and getters and setters are all important concepts in the realm of computer science. Classes can be used to define all of the various aspects of an object. Constructors help us initially set up attributes for an object. Getters and setters help us modify attributes.

Chapter 3: Getting Deeper with Class Objects

The very foundation of object oriented programming can ultimately be boiled down to classs and how they interact with each other. In object oriented programming, a class can be defined as a template that can be used to create a particular object. In other words, within a class, we are able define the various behaviors of the class as well as some of the attributes that may be associated with it.

For example, we can create a class called class_design_and_usage that will ultimately create a class, and then create two methods within the class itself. These two methods are called greet and serve.

```java
public class class_design_and_usage {
    private String name;
    private int age;
   // Constructor to initialize object properties
    public TennisPlayer(String name, int age) {
        this.name = name;
        this.age = age;
    }
   // two methods within this class, one to greet and one to show the serve
status.
    public void greet() {
        System.out.println("Hello, I'm " + name + "! Let's play tennis!");
    }

    public void serve() {
        System.out.println(name + " serves the ball with power!");
    }

    public static void main(String[] args) {
        TennisPlayer player1 = new TennisPlayer("Roger Federer", 39);
        TennisPlayer player2 = new TennisPlayer("Serena Williams", 39);

        player1.greet();
        player1.serve();

        player2.greet();
        player2.serve();
    }
}
```
Class_design_and_usage.java

There are many ways in which we are bale to make objects “do” things. The primary way in which we are able to make classes do things are through something called a method. A method is essentially a function that is contained within a class. In other words, we are able to define various behaviors, attributes, and logic of the class through methods.

Another important concept in the realm of object oriented programming is the coenpt of dynamic and static variables or functions. In the realm of computer science, Dynamic variables can be defined as a variable of an object that can actively be modified. In other words, within a class, an object will have a list of its own individual variables, and these variables can actively be changed for each individual object. In contrast, static variables can be defined as a variable that is assigned to the class itself and not the object of the class. In other words, each individual object will all contain the same variable.

For example, we can create a class called dynamic_and_static_variables. This particular class uses both dynamic and static variables. Court count will be a static variable and courtname will be a dynamic variable.

```java
public class dynamic_and_static_variables {
    private static int courtCount = 0;
    private String courtName;

    public TennisCourt(String courtName) {
        this.courtName = courtName;
        courtCount++;
    }

    public void displayCourtInfo() {
        System.out.println("Court Name: " + courtName);
        System.out.println("Number of Courts: " + courtCount);
    }

    public static void main(String[] args) {
        TennisCourt court1 = new TennisCourt("Centre Court");
        court1.displayCourtInfo();

        TennisCourt court2 = new TennisCourt("Court 1");
        court2.displayCourtInfo();

        TennisCourt court3 = new TennisCourt("Court 2");
        court3.displayCourtInfo();
    }
}
```

Dynamic_and_static_variables.java

There are many ways in which are able to approach class design. One way in which we can approach class design is through a concept called inheritance. In the realm of computer science, inheritance can be defined as a hierarchy of classes that will inherit various traits of a parent class. This will be explained in greater detail in Chapter 5. Another way in which we can approach class design is by making an effort to improve the reusability of classes. In other words, by keeping classes simple, we can find that there will be a number of different scenarios where using that particular class may be helpful.

Chapter 4: Testing and Exception Handling

Another concept in the realm of computer science is the concept of testing and exception handling. More specifically, test driven development is important to understand. Test driven development can be defined as the practice of writing tests first, before the actual code is written. The overall lifecycle of test driven development will begin with writing a test, next we will write the code that will satisfy the test. Third, we will then run the test to see if the code is written properly. Finally, we will refactor the code and make any additional changes that may improve the way in which it is architected.

Two important types of testing are Junit testing and driver testing. In the realm of computer science, Junit testing can be defined as a specific type of testing framework that provides functionality for a variety of different tests that can be used, such as annotations, assertions, and utilities. In contrast, driver testing is a method of testing that is performed on a much broader scope. Rather than testing each individual function or class like a Junit test may do, driver testing largely focuses on the broad scale of things and how all of the classes will interact with each other and if there are any issues with the way in which each of the individual classes will communicate with each other.

Another important concept in the realm of computer science is the concept of exception handling and how it can help us create better working code. In the realm of computer science, exception handling can be defined as the process of anticipating possible areas in which the code can break, and writing additional sets of instructions, or exceptions, in the event that a particular error has occurred within the code that we are writing.

For example, we can create a class called unit_testing_and_exception_handling that will perform a series of tests in order to ensure that the code is working properly. We can also include an exception for when an invalid score is inputted into the system. In the event that this will happen, rather than the entire program breaking, a message will print to the user telling them that an invalid score was entered.

```java
import org.junit.Test;
import static org.junit.Assert.*;


public class unit_testing_and_exception_handling {
    @Test
    public void testGetScore() {
        TennisGame game = new TennisGame("Player A", "Player B");

        // Test initial score
        assertEquals("Love - Love", game.getScore());

        // Test player 1 scores
        game.player1Scores();
        assertEquals("15 - Love", game.getScore());
        // Test player 2 scores
        game.player2Scores();
        assertEquals("15 - 15", game.getScore());

        // Test exception handling for invalid score
        try {
            game.setScore(-1, 0);
            fail("Exception not thrown for invalid score");
        } catch (IllegalArgumentException e) {
            assertEquals("Invalid score value", e.getMessage());
        }
    }
}
```
Unit_testing_and_exception_handling.java

There are many benefits of well designed exceptions. One benefit of well designed exception handling is the fact that there will be a limited number of instances in which the application will break. This is particularly helpful, especially when the program is presented to the user. The user will have a much better experience if the application is able to pick up on various instances in which the code might break, and correct itself accordingly. Another benefit of well designed exception handling is the readability and maintainability of the code that is written. In other words, if there are a large number of well architected exceptions written into an application, it can make debugging a much better experience for the developer. Rather than trying to deconstruct the code and figure out where the origin of an error is, well written exceptions will accurately and efficiently describe the issue at hand and how to fix it. This can be very helpful, especially when working with a team of developers, or passing code on to another new developer. As can be seen with these two examples, well architected exception handling can be beneficial for not only the user, but also the developer.

There are many different reasons as to why we should implement exceptions. Testing exception handling is largely focused on ensuring that in various instances that break the application, the exception is properly caught. Not only is it important to test that the exception is properly caught, but it is also important to ensure that the exception handles and corrects itself accordingly.

In conclusion, there are many important concepts in the realm of computer science when it comes to testing and exception handling. Test driven development is an important practice in the industry as it allows for developers to define the way in which a particular program should behave and test before it is actually written. Additionally, Junit testing and driver testing are both important types of tests that can be used to ensure that an application is working properly. Thirdly, exception handling is important to limit the amount of opportunity an application may have from failing, by anticipating various instances in which an application could potentially fail, and writing code to remediate the issue if it were to come up. Fourthly, well designed exception handling is beneficial not only for the overall user experience but also for developers as they continue to build and develop the application.

Chapter 5: Inheritance and Composition

Inheritance and composition are both important concepts in the realm of computer science. Inheritance can be defined as the concept of a class inheriting various methods from another class. Furthermore, the class that inherits the methods is called the subclass. This is particularly important because it allows us to reuse code. Composition can be defined as a strategy in object oriented programming in which a class is composed of other classes. This is important because it allows us to create a complex concept with the combination of various classes. Through the use of this chaining, we are able to create incredibly complex concepts within our code.

For example, let’s say for instance that we create a class called inheritance. We first start off by defining the TennisPro class, which is a subclass of TennisPlayer. In other words, the TennisPro class will inherit everything from the TennisPlayer class. In this particular instance, the TennisPro class will inherit the play() method, and add its own serve() method on top of that.

```java
public class inheritance {
    private String name;

    public void TennisPlayer(String name) {
        this.name = name;
    }

    public void play() {
        System.out.println(name + " is playing tennis.");
    }

    public static void main(String[] args) {
    }
}

public class TennisPro extends TennisPlayer {
    private int ranking;

    public TennisPro(String name, int ranking) {
        super(name);
        this.ranking = ranking;
    }

    public void serve() {
        System.out.println(getName() + " serves with precision.");
    }
    public int getRanking() {
        return ranking;
    }
}

class Main {
    public static void main(String[] args) {
        TennisPro player = new TennisPro("Roger Federer", 3);
        player.play();
        player.serve();
        System.out.println(player.getName() + "'s ranking: " +
player.getRanking());
    }
}
```

Inheritance.java

There are a number of different reasons why program designers may choose to implement inheritance. One instance in which it may be better to use inheritance over composition is if the subclass happens to be a specialized type of the superclass.

There are also a number of different reasons why program designers may choose to implement composition. Alternatively, it may be better to use composition if the class is composed of a combination of other classes.

For example, let's say for instance that we create a class called composition. In this particular instance, we have a TennisPlayerWithRacket class. We essentially have two private variables, player of type TennisPlayer and racket of type TennisRacket. Since the two objects have instance variables, we can say that the TennisPlayerWithRacket class is composed of the TennisPlayer and TennisRacket classes.

```java
public class composition {
    private String name;

    public TennisPlayer(String name) {
        this.name = name;
    }

    public void play() {
        System.out.println(name + " is playing tennis.");
    }

    public static void main(String[] args) {
    }
}
```

```
public class TennisRacket {
    private String brand;

    public TennisRacket(String brand) {
        this.brand = brand;
    }

    public void swing() {
        System.out.println("Swinging the " + brand + " racket.");
    }
}

public class TennisPlayerWithRacket {
    private TennisPlayer player;
    private TennisRacket racket;

    public TennisPlayerWithRacket(String playerName, String racketBrand) {
        player = new TennisPlayer(playerName);
        racket = new TennisRacket(racketBrand);
    }

    public void playTennis() {
        player.play();
        racket.swing();
    }
}

class Main {
    public static void main(String[] args) {
        TennisPlayerWithRacket player = new TennisPlayerWithRacket("Roger
Federer", "Wilson");
        player.playTennis();
    }
}
```

Composition.java

Dynamic dispatching is another important concept in the realm of computer science. In the realm of computer science, dynamic dispatching can be defined as the practice of utilizing different classes that are all related through inheritance, and treating them as if they are all objects of the inherited superclass.

Another important concept in the realm of computer science is abstract and concrete classes. Abstract classes can be defined as a class that behaves as if it were a blueprint for other classes. Abstract classes will also contain abstract methods that may be visible and may exist, but are not actually implemented within the class itself. In contrast, a concrete class is a class that not only contains all of the methods, but they are also implemented as well.

Chapter 6: Interfaces

Interfaces is another important concept to know in the realm of object oriented programming. An interface can be defined as a way in which we can define a group of methods that are implemented by a class. In other words, it is a way in which we can define a blueprint for other classes to follow.

For example, we can create an interface called Interface, which is declared using the interface keyword at the top of the block of code. In this particular situation, both the ProfessionalPlayer class and the AmateurPlayer class are both implementing the TennisPlayer interface. This means that they use the playTennis() method.

```
interface Interface {
    void playTennis();
}

class ProfessionalPlayer implements TennisPlayer {
    private String name;

    public ProfessionalPlayer(String name) {
        this.name = name;
    }

    @Override
    public void playTennis() {
        System.out.println(name + " is playing tennis as a professional 
player.");
    }
}

class AmateurPlayer implements TennisPlayer {
    private String name;

    public AmateurPlayer(String name) {
        this.name = name;
    }

    @Override
    public void playTennis() {
        System.out.println(name + " is playing tennis as an amateur player.");
    }
}

class Main {
    public static void main(String[] args) {
        TennisPlayer proPlayer = new ProfessionalPlayer("Roger Federer");
        proPlayer.playTennis();

        TennisPlayer amateurPlayer = new AmateurPlayer("John Doe");
        amateurPlayer.playTennis();
    }
}
```

Interface.java

There are a few differences between an abstract class and an interface. One difference between an abstract class and an interface is the fact that it is possible for an abstract class to implement only part of the methods. It is also possible for an abstract class to implement other things such as constructors. Alternatively, an interface is only able to define methods, without actually implementing them.

There are certainly a few reasons why we may want to use an interface over an abstract class and vice versa. One reason why we may want to use an interface over an abstract class is if we are looking to achieve inheritance multiple times. This is particularly helpful because a class can implement multiple interaces, where there can only be done once with an abstract class. One reason why we may want to use an abstract class over an interface is if we are looking to create a class that has an implementation right off that bat. In other words, if we are looking to have a default implementation, an abstract may be the best choice in this particular instance.

There are certainly a number of reasons why there are restrictions placed on an interface. One reason as to why there are restrictions that are placed on an interface is to ensure that they are used in the way in which they were intended. An example of this is the fact that it cannot have a private modifier. With that being said, it is a way to ensure that classes that inherit this code will have the proper access. In other words, these restriction ensure that there is abstraction and modularity in the code, which ultimately allows for the code to be in more of a reusable state.

Chapter 7: The Dangers of Inheritance

It is also important to learn about the negative side of inheritance and why it might not be the best option in various circumstances. There are many reasons and situations which may lead to inheritance not being the best option. One major problem that may arise in the world of inheritance is when we are working on code, and the number of classes may start to become overwhelming, messy, and sloppy. In other words, we can have an incredibly deep hierarchy of classes, but this may turn out to be overly complex and lead to a number of other issues. One issue that may arise is if we have a code that has five or six layers of inheritance, if we change, for example, the second layer of inheritance, this could create a domino effect in our code. This can be incredibly detrimental to the overall productivity of your team.

There are a number of ways in which we are able to avoid these problems. One way in which we can avoid some of the dangers of inheritance is through the use of composition, when inheritance isn't necessary. Since composition reduces the complexity between classes, it will make the code seem to be not as bloated, and will ultimately be less of a problem when trying to change a class that is at the bottom of the food chain.

For example, let's say for instance that we have a class called dangers_of_inheritance. In this instance we have a base class called TennisPlayer. We then have a constructor that takes a name as a parameter. We then have a series of classes that are inheriting eachother. We have ProfessionalPlayer and SuperstarPlayer. This can create potential problems, because let's say for instance that in the future we would like to modify the playTennis() method, this will impact not only ProfessionalPlayer but also SuperstarPlayer. With that being said, we need to be careful when modifying code that involves inheritance, because it can cause a domino effect of problems if we aren't careful.

```
public class dangers_of_inheritance {
   private String name;

    public TennisPlayer(String name) {
        this.name = name;
    }

    public void playTennis() {
        System.out.println(name + " is playing tennis.");
    }

    public static void main(String[] args) {
    }
}

class ProfessionalPlayer extends TennisPlayer {
    public ProfessionalPlayer(String name) {
        super(name);
    }

    public void playTennis() {
        System.out.println("Playing tennis like a professional.");
    }
```

```java
}

class SuperstarPlayer extends ProfessionalPlayer {
    public SuperstarPlayer(String name) {
        super(name);
    }

    public void playTennis() {
        System.out.println("Playing tennis like a superstar.");
    }
}

class Main {
    public static void main(String[] args) {
        TennisPlayer player = new SuperstarPlayer("Roger Federer");
        player.playTennis();
    }
}
```

Dangers_of_inheritance.java

Chapter 7: Lists, Arrays, and Other Data Structures

One important concept in the realm of computer science is the concept of a list. Lists can be an incredibly important tool to use because it allows us to create, store, edit, and delete various pieces of information or data. A list can be defined as a type of data structure that will store data in a specific order. This data, otherwise known as elements, can be inserted, deleted, and retrieved. For example, let's say for instance we are looking to make a list of tennis courts that are available to play on. In order to do this, we can create a list of integers, where each integer resembles the number of the court.

```java
public class list {
    public static void main(String[] args) {
        List<Integer> availableCourts = new ArrayList<>();
        availableCourts.add(1);
        availableCourts.add(2);
        availableCourts.add(3);
        availableCourts.add(4);
        availableCourts.add(5);

        int bookedCourt = 3;
        if (availableCourts.contains(bookedCourt)) {
            availableCourts.remove(Integer.valueOf(bookedCourt));
            System.out.println("Court " + bookedCourt + " has been booked.");
        } else {
            System.out.println("Court " + bookedCourt + " is not available.");
        }

        int freedCourt = 2;
        if (!availableCourts.contains(freedCourt)) {
            availableCourts.add(freedCourt);
            System.out.println("Court " + freedCourt + " is now available.");
        } else {
            System.out.println("Court " + freedCourt + " is already available.");
        }

        System.out.println("Currently available courts: " + availableCourts);
    }

}
```

Another important concept in the realm of computer science is filtering a list. The ability to filter a list is important because it allows us to take a certain list and extract various pieces of information from that list. Filtering a list can be defined as the practice of taking a list and extracting certain elements from within the list that fit some type of criteria. For example, let's say for instance that we are looking to take the list of tennis courts that are available to play on, but we want to filter only the first five courts that are available.

```
public class filter {
    public static void main(String[] args) {
        List<Integer> availableCourts = new ArrayList<>();
        availableCourts.add(1);
        availableCourts.add(2);
        availableCourts.add(3);
        availableCourts.add(4);
        availableCourts.add(5);
        availableCourts.add(6);
        availableCourts.add(7);
        availableCourts.add(8);
        availableCourts.add(9);
        availableCourts.add(10);


        // Display only the first five courts
        List<Integer> firstFiveCourts = availableCourts.subList(0, Math.min(5,
availableCourts.size()));

        System.out.println("First five available courts: " + firstFiveCourts);
    }
}
```

Another important concept is the concept of sorting a list. This is important because it allows us to take a list and order the elements within the list in a way that can be more useful to us. Sorting a list can be defined as the practice of taking a list and sorting all of the elements from within the list in order to fit some type of criteria. For example, let's say for instance that we are looking to take the list of tennis courts that are available to play on, but we want to sort the tennis courts from the largest number to the smallest number.

```
public class filter {
    public static void main(String[] args) {
        List<Integer> availableCourts = new ArrayList<>();
        availableCourts.add(7);
        availableCourts.add(2);
        availableCourts.add(10);
        availableCourts.add(1);
        availableCourts.add(5);
        availableCourts.add(9);
        availableCourts.add(8);
        availableCourts.add(4);
        availableCourts.add(6);
        availableCourts.add(3);

        // Sort the list in descending order
        Collections.sort(availableCourts, Collections.reverseOrder());

        // Display only the first five courts
        List<Integer> firstFiveCourts = availableCourts.subList(0, Math.min(5,
availableCourts.size()));

        System.out.println("First five available courts (sorted): " +
firstFiveCourts);
    }
}
```

Another concept that is worth noting is the concept of an interface. This is important because it allows us to create a program where different classes can have the same methods, while simultaneously keeping their own implementation. An Interface can be defined as a set of instructions that specifies various methods that a class will have to implement and use. For example, we can have an interface "TennisPlayer" that contains two methods playTennis() and watchTennis(). We can also make two classes that will use this interface. We can have “Player1” and “Player2” as the classes. With that being said, both classes can have its own implementation of interface.

```java
// Define the TennisPlayer interface
interface TennisPlayer {
    void playTennis();
    void watchTennis();
}

// Implement the interface in Player1 class
class Player1 implements TennisPlayer {
    @Override
    public void playTennis() {
        System.out.println("Player1 is playing tennis.");
    }

    @Override
    public void watchTennis() {
        System.out.println("Player1 is watching tennis.");
    }
}

// Implement the interface in Player2 class
class Player2 implements TennisPlayer {
    @Override
    public void playTennis() {
        System.out.println("Player2 is playing tennis.");
    }

    @Override
    public void watchTennis() {
        System.out.println("Player2 is watching tennis.");
    }
}
public class interface {
    public static void main(String[] args) {
        TennisPlayer player1 = new Player1();
        TennisPlayer player2 = new Player2();

        player1.playTennis();
        player1.watchTennis();

        player2.playTennis();
        player2.watchTennis();
}
```

Another important concept is the concept of a linked list. This is an important concept because it allows us to access data in a certain way, and is one of the fundamental concepts of a data structure. A linked list can be defined as a type of data structure where each element will resemble a node. And each node will point to the next node in the list. Essentially, this will create a chain of nodes that are all connected to one another. For example, let's say for instance that we have a tennis tournament, and we are looking to access data for each specific match. The catch is that we don't need to be able to access the data randomly.. In fact, we only need to access the data when it is time for the players to play the match, based on a predetermined schedule. With all of that being said, we can use the concept of a linked list, where each element or node will contain information about the match, and that specific node will then point to the next node, which will be the next match that is queued up to play.

An array is also an important concept to learn and understand. This is because it is also another type of data structure that can be used to access data in a certain way. An array can be defined as a type of data structure where each element is assigned an index, and each individual element can be accessed through the use of the index. For example, let's say for instance that we are looking to make a list of all the tennis players in the tournament, and we need to be able to access any of the tennis players on command. We are able to achieve this through the use of an array, where we assign an index to each tennis player. We can then use the index to find the tennis player.

```java
class TennisPlayer {
    String name;
    int rank;

    public TennisPlayer(String name, int rank) {
        this.name = name;
        this.rank = rank;
    }

    public void playMatch() {
        System.out.println(name + " is playing a match.");
        // Implement the logic to play the match here
    }

    public void watchMatch() {
        System.out.println(name + " is watching a match.");
        // Implement the logic to watch the match here
    }
}

public class array {
    public static void main(String[] args) {
        TennisPlayer[] players = new TennisPlayer[8]; // Array to store 8 players

        players[0] = new TennisPlayer("PlayerA", 1);
        players[1] = new TennisPlayer("PlayerB", 2);
        players[2] = new TennisPlayer("PlayerC", 3);
        players[3] = new TennisPlayer("PlayerD", 4);
        players[4] = new TennisPlayer("PlayerE", 5);
        players[5] = new TennisPlayer("PlayerF", 6);
        players[6] = new TennisPlayer("PlayerG", 7);
        players[7] = new TennisPlayer("PlayerH", 8);

        // Accessing players by index
        int playerIndex = 3; // For example, access PlayerD
        TennisPlayer player = players[playerIndex];
        System.out.println("Accessed player: " + player.name);

        // Example: Playing a match
        player.playMatch();

        // Example: Watching a match
        player.watchMatch();
    }
}
```

Chapter 8: Recursive Design

Recursive design is important in the realm of computer science because it allows for us to solve a particular problem by breaking it down into smaller pieces. Recursive design can be defined as a strategy in computer science used to solve a particular problem by breaking it down into smaller subproblems over and over again until the overall problem has been solved.

Recursive design can consist of for each loops and counting for loops. Using traditional for each and counting for loops are another important concept in the realm of computer science because they allow us to loop through a particular dataset without actually having to worry about how many elements are actually in the dataset. For each loops can be defined as a strategy in computer science, similar to recursive design, where we are looking to solve a particular problem by breaking it down into smaller subproblems, however we do not have to worry about specifying a certain amount of times the program will loop through, as it will only loop through for the exact amount of items that are within the dataset.

In contrast, a counting for loop can be defined as a strategy in computer science where we are looking to solve a particular problem by breaking it down into smaller subproblems, but rather than having the amount of times it will loop through be defined by the amount of items in a list, it will be defined by a specific number. For example, let's say for instance that we are writing a program for a ball machine. We can tell the ball machine to shoot exactly 20 balls. We can do this by writing a for loop, which will count the amount of times that it has iterated through the loop, and will stop once it reaches the designated number.

```
public class BallMachine {
    public static void main(String[] args) {
        int numberOfBallsToShoot = 20;

        for (int i = 1; i <= numberOfBallsToShoot; i++) {
            System.out.println("Shooting ball " + i);

        }

        System.out.println("All balls have been shot.");
    }
}
```

Chapter 9: Equality and Basic Sorting Algorithms

Equality is an important concept in the realm of computer science because it allows us to determine whether or not two particular objects are the same. Equality can be defined as the practice of comparing two objects of different types and determining whether they contain the same values. For example, let's say for instance we have a tennis match who's scores were recorded in two separate datasets. We want to write a block of code that will confirm that both of the scores are equal to each other, and that there is no inconsistency.

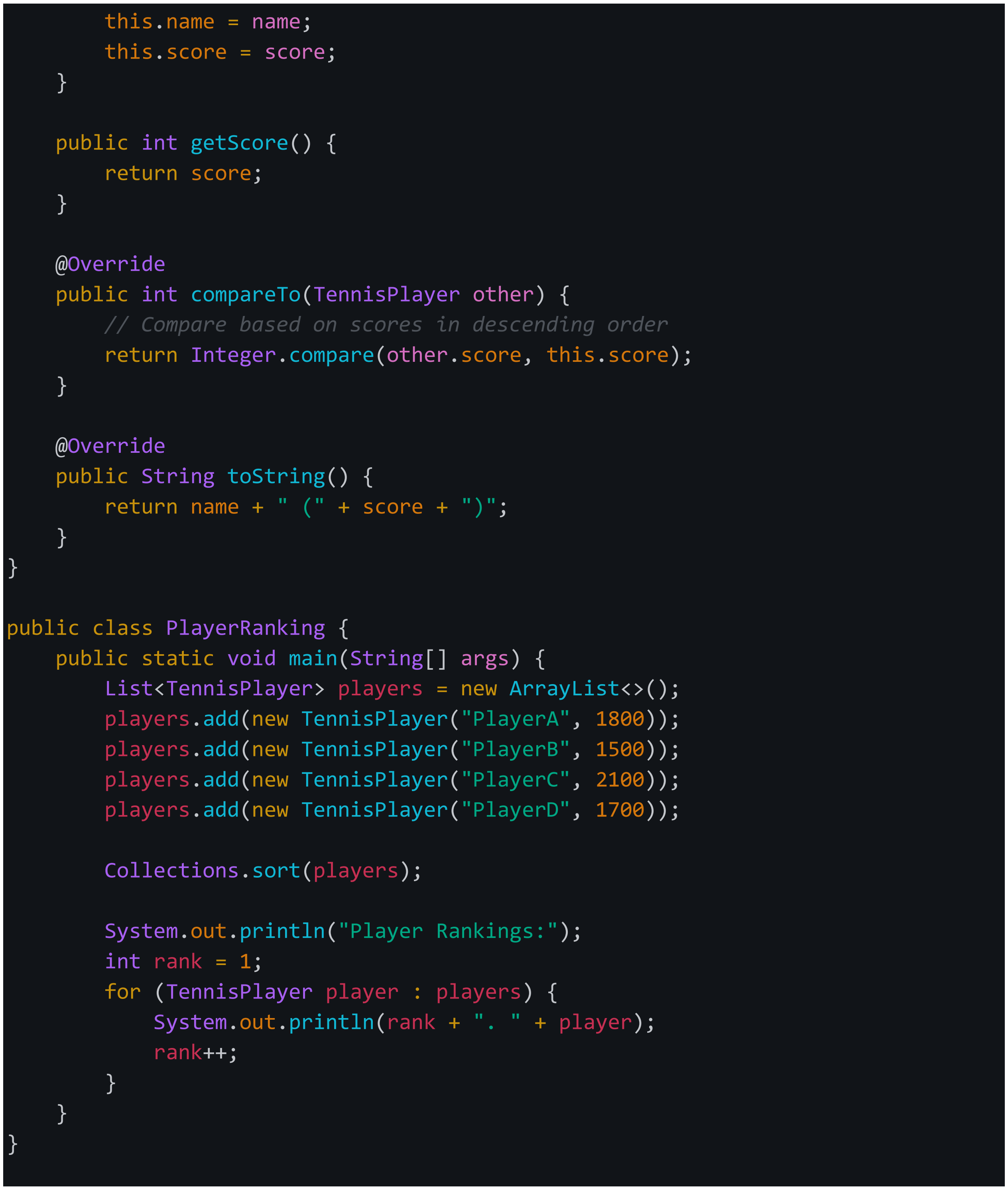
One type of equality is called equality of simple objects. This is when we take a look at two different objects, and determine whether or not the values of the two objects are equal to each other. Equality of simple objects can be defined as the practice of comparing two objects and what their values are in order to determine whether or not they are the same. For example, let's say that we have two strings, which are equal to the names of two different tennis players. We can use the equals() method to determine if the names of the two tennis players are the same.

```
public class TennisScoreComparison {
    public static void main(String[] args) {
        int[] dataset1 = {6, 3, 7, 6};
        int[] dataset2 = {6, 3, 7, 6};

        boolean scoresConsistent = true;
        if (dataset1.length != dataset2.length) {
            scoresConsistent = false;
        } else {
            for (int i = 0; i < dataset1.length; i++) {
                if (dataset1[i] != dataset2[i]) {
                    scoresConsistent = false;
                    break;
                }
            }
        }

        if (scoresConsistent) {
            System.out.println("Scores are consistent.");
        } else {
            System.out.println("Scores are inconsistent.");
        }
    }
}
```

Equality for subtypes and double dispatch are also important concepts. This is because it is important to be able to compare not only values, but also classes. Equality for subtypes and double dispatch are also important concepts. This is because it is important to be able to compare not only values, but also classes. As for equality for subtypes, let's say for instance that we have a class called tennisRacket. Let's also say that we have two classes named Wilson and Babolat that both extend racket. We are able to use .equals to determine whether or not a particular class is equal to another class. As for double dispatch, we can do the same thing where we compare two classes, but rather than looking to see if they are equal, we will look to determine the intersection between the two classes.

Ordering is also an important concept in the realm of computer science, because it allows us to sort objects in a certain way that can be useful to us. Ordering can be defined as the practice of reorganizing the objects in such a way such that it fits some type of criteria or makes the new order useful in some way. For example, let's say for instance that we have a list of tennis players and all of their scores. We want to order the list in such a way such that the person with the highest score will be ranked number one, and the person with the lowest score will be ranked last, and so forth. We can do this by using a comparable interface.

```
class TennisPlayer implements Comparable<TennisPlayer> {
    private String name;
    private int score;

    public TennisPlayer(String name, int score) {
```



```java
        this.name = name;
        this.score = score;
    }

    public int getScore() {
        return score;
    }

    @Override
    public int compareTo(TennisPlayer other) {
        // Compare based on scores in descending order
        return Integer.compare(other.score, this.score);
    }

    @Override
    public String toString() {
        return name + " (" + score + ")";
    }
}

public class PlayerRanking {
    public static void main(String[] args) {
        List<TennisPlayer> players = new ArrayList<>();
        players.add(new TennisPlayer("PlayerA", 1800));
        players.add(new TennisPlayer("PlayerB", 1500));
        players.add(new TennisPlayer("PlayerC", 2100));
        players.add(new TennisPlayer("PlayerD", 1700));

        Collections.sort(players);

        System.out.println("Player Rankings:");
        int rank = 1;
        for (TennisPlayer player : players) {
            System.out.println(rank + ". " + player);
            rank++;
        }
    }
}
```

Linear search, binary search, simple sorting, and smart sorting are all important concepts in the realm of computer science because they are all various ways in which we can search through data in order to find a particular element. Linear search can be defined as a type of searching algorithm that will search through each element one by one until it has found a certain specified value. For example, let's say for instance we have a tennis match who's scores were recorded in two separate datasets. We want to write a block of code that will confirm that both of the scores are equal to each other, and that there is no inconsistency.

Binary search can be defined as a type of searching algorithm where it will eliminate half of the possibilities at each step, until it has found the element that it is looking for. Let's say for instance we have the exact same problem. Instead of checking each value one by one, we can use binary search to divide the load in half until we are able to eventually find the value 22.

Simple sorting can be defined as a type of sorting algorithm that will sort elements based on specific criteria. For example, let's say for instance you have the list of tennis player rankings, but they are scrambled in an unorganized fashion. We can use something like simple sorting to sort all of the rankings from smallest to largest.

```
public class PlayerRankingSorting {
    public static void main(String[] args) {
        int[] rankings = {5, 2, 9, 1, 7, 3, 8, 4, 6};

        bubbleSort(rankings);

        System.out.println("Sorted Rankings: " + Arrays.toString(rankings));
    }

    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;

            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {

                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }

            // If no two elements were swapped in the inner loop, the array is
already sorted
            if (!swapped) {
                break;
            }
        }
    }
}
```

Smart sorting consists of more complex algorithms that are better for time complexity. For example, we can have the same scrambled list of tennis player rankings, but we can use something like the quick sort algorithm which will use a divide and conquer strategy to sort the rankings from smallest to largest.

Chapter 10: Hierarchical Data Representations

Hierarchical organization of data is an important concept in the realm of computer science because it allows us to interpret various pieces of data in a way that shows structure and organization.

One type of hierarchical organization is data representation in a hierarchy. Data representation is an important concept in the realm of computer science because it allows us to create various relationships between nodes. Hierarchical organization of data can be defined as creating and organizing data in such a way where there are parent and child relationships between all of the nodes. For example, let’s say we are writing a program that keeps track of all of the achievements that a player can achieve. We want to keep track of who wins the entire year. In order to do this, we can have a hierarchy of achievements, such as match, title, grand slam, year. This way, there is a parent child relationship between each of these achievements.

```
class Achievement {
    private String name;

    public Achievement(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void display() {
        System.out.println(name);
    }
}

class MatchAchievement extends Achievement {
    public MatchAchievement(String name) {
        super(name);
    }
}

class TitleAchievement extends Achievement {
    public TitleAchievement(String name) {
        super(name);
    }
}

class GrandSlamAchievement extends TitleAchievement {
    public GrandSlamAchievement(String name) {
        super(name);
    }
}

class YearAchievement extends GrandSlamAchievement {
    public YearAchievement(String name) {
        super(name);
    }
}

public class PlayerAchievements {
    public static void main(String[] args) {
        Achievement match = new MatchAchievement("Won a match");
        Achievement title = new TitleAchievement("Won a title");
        Achievement grandSlam = new GrandSlamAchievement("Won a Grand Slam");
        Achievement year = new YearAchievement("Player of the Year");

        match.display();
        title.display();
        grandSlam.display();
        year.display();
    }
}
```

Creating and growing a hierarchy is another important concept to understand because there is a specific method that needs to be followed when we are looking to add elements to an already existing data structure. For example, we can have the same program that tracks the achievements of various tennis players. In this case, we can create an instance where we are able to create new tennis players. By adding a new player, this particular player will automatically inherit all of the other classes and methods that are associated with being a tennis player.

Counting data in a hierarchy is another important concept in the realm of computer science because it allows us to determine the overall shape of a particular hierarchy. Counting data in a hierarchy can be defined as the process of determining how many elements are within a particular category or section of a hierarchical structure. For example, let’s say for instance that we are continuing the program that will determine who is ranked what. We can count the data that is in our hierarchy in various ways. For example, we can count the number of matches won by a getTotalMatch() method. This will traverse through all of the matches and count the number of matches won in each round.

Chapter 11: Model View Controller Architecture

The concept of a model in model controller view architecture is important because it allows us to manage all of the data from the application. The model can be defined as the logic that handles all of the data in a program. For example, let's say for instance that we are building a program that will manage all of the accounts for the tennis players. This particular program allows us to add, delete, and edit the profiles. In this particular instance, the model part of the program would contain all of the logic for how these particular profiles are added, deleted, and edited.

The concept of the view portion of model controller view architecture is important because it allows us to manage exactly what the user sees. The view can be defined as the part of the program that is responsible for everything that the user sees. For example, let's say for instance that we are building the same tennis player profile management system. The view portion of the program will contain all of the logic for what the user will actually see.

The concept of the controller portion of model controller view architecture is important because it allows us to determine how the user is actually able to use the app and how it will behave depending on the various inputs and outputs. The controller can be defined as the part of a program that contains all of the logic for how a particular program will react and behave depending on how the user interacts with it. For example, let's say that we are again building the same tennis player profile management system. The controller portion of the program would contain all of the logic that will depict what will actually happen when a particular button is pressed, or when anything else in the program is pressed.

Chapter 12: Object-Oriented Design Principles Revisited

Iterator design pattern is an important concept in the realm of computer science because it allows for us to be able to access elements that may be within a particular data structure, without showing any logic or design of the data structure. Iterator Design pattern can be defined as a type of strategy which will iterate through various elements without revealing how the elements are actually connected to each other. For example, let's say for instance that we have a list of all the tennis players in a particular state. We can use an iterator design pattern to iterate through this list, without revealing exactly how this list is constructed or how the algorithm traverses through a dataset.

The strategy pattern is an important concept as well because it allows the user to decide what may be the best algorithm to use. Strategy pattern can be defined as a strategy that provides multiple different algorithms for multiple different scenarios, and allows the user to decide which strategy may be the best option. For example, let’s say for instance that we have a tennis player that is looking to sign up for a tournament, and the tennis player has the option to choose how he or she would like to sign up for the tournament. The program will change which algorithm to run depending on whether the user would like to use paypal, credit card, or cash.

Adapters are another important concept in the realm of computer science because it allows two incompatible interfaces to work together. Adapters can be defined as a specific design in which will take the interface of one class and make it compatible with the interface of another class. For example, let’s say for instance that we have data that contains one state of tennis players and another piece of data that contains another state of tennis players, but the format of the data is different and incompatible. We can use an adapter to convert the data into either format so these two datasets can work with each other.

References

w3schools.com. (n.d.). Java Tutorial. Retrieved from https://www.w3schools.com/java/

w3schools.com. (n.d.). Java Data Types. Retrieved from https://www.w3schools.com/java/java_data_types.asp

w3schools.com. (n.d.). Java Inheritance. Retrieved from https://www.w3schools.com/java/java_inheritance.asp