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.

```java
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.

```java
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.

```
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.

```
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.

```
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.");
    }
```

```
}

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

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