Part 1 OO Questions

1. (5 points) *What are three ways modern Java interfaces differ from a traditional OO interface design that describes only function signatures and return types to be implemented? Provide a Java code example of each.*

Java introduced several new features for interfaces that include support for things such as generics, default methods, static methods, private methods, nested classes and interfaces, nested enums and others. Below are code examples of 3 of the new supported features that use default, static and private methods along with a simple explanation as to why you would use these features.

Default method

Default methods defined in an interface do not need to be explicitly overridden in the class that implements the interface. They are made available as default. The example below shows an ATM interface with an object type WalkUpATM that implements the interface. The ATM can have two default methods made available to the interface.

One of the main reasons Java supported default methods is to provide API stability to classes implementing a previous interface. If you have already defined an interface, and many classes have already implemented it, you would need to add a method to every class that inherits from that interface. This provides a way to add in a method w/o having to explicitly define it.

```
public interface ATM {

  String getBankName();

  default boolean withdraw(Double money)  {

    System.out.println("Withdrawing: " + money);
    return true;
  }

  default boolean deposit(Double money)  {

  System.out.println("Depositing: " + money);
  return true;
  }
}

public class WalkUpATM implements ATM {

  private String bankName;

  WalkUpATM(String bankName){
    this.bankName = bankName;
```

```
  }

  @Override
  public String getBankName(){
    return bankName;
  }
}

public static void main(String[] args) {

    Double money = 100.00;

    ATM walkupATM = new WalkUpATM("WellsFargo");
    System.out.println(walkupATM.getBankName());  // #> WellsFargo
    walkupATM.deposit(money);  // #> Depositing: 100.0
    walkupATM.withdraw(money);  // #> Withdrawing 100.0
  }
```

Static method

You can define a static method in an interface like you can in a class. Using the same example as above, the ATM interface shows the use of a static method. In this example, you can define something that is common that all ATMs that might need, such as an encoding format that isn't part of an ATM object, just a static encoding format. Java lets you define this within an interface.

```
public interface ATM {

  String getBankName();

  default boolean withdraw(Double money)  {

    System.out.println("Depositing: " + money);    return true;
  }


  default boolean deposit(Double money)  {

  System.out.println("Depositing: " + money);
  return true;
  }

  static public String getMagStripeEncoding()  {

    String encoding =  "Some mag stripe encoding format to use here";
    return encoding;

  }

}


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

```
  Double money = 100.00;

  ATM walkupATM = new WalkUpATM("WellsFargo");
  System.out.println(walkupATM.getBankName());  // #> WellsFargo
  walkupATM.deposit(money);  // #> Depositing: 100.0
  walkupATM.withdraw(money);  // #> Withdrawing 100.0
  System.out.println(ATM.getMagStripeEncoding()); // #> Some mag stripe encoding format to use here
}
```

Private Methods

Java now supports private methods in an interface. This allows you to set up helper type methods that only methods in the interface can access. Stemming from the previous example, if you supported multiple magStripeformats and you wanted to break these out into functions, you could declare two helper methods to support two formats, v1 and v2, and make them a simple private method that only getMagStripeEncoding uses within the interface. This is all hidden from the consumer and the use here is more for code organization/readability.

```
public interface ATM {

 String getBankName();

 default boolean withdraw(Double money)  {

  System.out.println("Depositing: " + money);
  return true;
 }

 default boolean deposit(Double money)  {

 System.out.println("Depositing: " + money);
 return true;
 }

 static public String getMagStripeEncoding(int formatVersion)  {

  if (formatVersion > 1) {

     return getMagStripeEncodingV2();

  }
  else {
     return getMagStripeEncodingV1();
  }

 }

 private static String getMagStripeEncodingV2() {

  return "This is encoding for V2";
```

```
  }

  private static String getMagStripeEncodingV1() {

    return "This is encoding for V1";

  }
}

public static void main(String[] args) {

    Double money = 100.00;

   ATM walkupATM = new WalkUpATM("WellsFargo");
    System.out.println(walkupATM.getBankName());  // #> WellsFargo
    walkupATM.deposit(money);  // #> Depositing: 100.0
    walkupATM.withdraw(money);  // #> Withdrawing 100.0
    System.out.println(ATM.getMagStripeEncoding(1)); // #> This is encoding for V1
  }
```

2. (5 points) *Describe the differences and relationship between abstraction and encapsulation. Provide a Java code example that illustrates the difference.*

Abstraction

Abstraction refers to abstracting away details and providing only the essential information. In OO, abstraction can refer to an abstract class that provides common class members and methods to the subclasses. Take for example, an Bike class:

```
//abstract class
abstract class Bike{
   //abstract method
   public abstract void pedal() {
     //common code that makes a bike move by pedaling
   }
}

//MountainBike class extends Bike class
public class MountainBike extends Bike{

   public static void main(String args[]){
          Bike obj = new MountainBike();
          // abstract method here. Can override.
          obj.pedal();
   }
}
```

Common methods that all bikes have such as pedal() would exist in the base abstract class as this means less code to maintain vs. having every class potentially having the same code. I think a great way of explaining the concept of an Abstract class in OO is to show what it would look like w/o having Abstract class and the amount of redundant code that would be required to maintain it vs using an Abstract base class.

Encapsulation

Encapsulation is related to abstraction in that it hides away implementation details from the consumer but is a more general term to the code that is used to implement a method/function implementation that is hidden from the consumer. For instance, in java to make a rest http request, you can do simply do the following:

```
URL url = new URL("http://myfavoritesite,com/awesomeendpoint ");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("GET");
```

In this example above, all the user needs to do is use the URL class, set an URL, create the connection, and then make a GET request using method setRequestMethod. All the networking details are encapsulated in the URL classes that are hidden from the consumer.