Name _____ Period _____

Periodic Table

Your Tasks (Mark these off as you go)

- ☐ Define key vocabulary
- ☐ Describe the purpose of abstraction
- ☐ Differentiate between an abstract class and an interfaces
- ☐ Receive credit for this lab guide

☐ Define key vocabulary

Abstract class

Abstract method

Concrete class

Interface (as it applies to Java)

Override

Polymorphism

☐ Describe the purpose of abstraction

So far, we have only dealt with concrete classes. A *concrete class* is a class that has an implementation for all of its methods. They cannot have any unimplemented methods.

Abstract classes are just like concrete classes, except for two differences. First, abstract classes allow for the inclusion of unimplemented methods. Second, objects of abstract classes cannot be instantiated. The subclass

which inherits the abstract class is responsible for implementing the unimplemented methods and instantiating the abstract class. This process of hiding details and showing only the essential information to the user is referred to as abstraction.

The abstract class below is intended to model an element and its isotopes. It is declared with the key word *abstract* because it contains abstract methods. The abstract methods, *addIsotope, getIsotopes, and getAvgAtomicMass* have no implementation. They must, however, be implemented by the class that inherits it.

Element

```
public abstract class Element
{
private String name;
private String symbol;
private int atomicNumber;

    public Element(String n, String s, int an){
        name = n;
        symbol = s;
        atomicNumber = an;
    }

    public String getName () {
       return name;
    }

    public String getSymbol () {
       return Symbol;
    }

    public int getNumber () {
       return atomicNumber;
    }

    public abstract void addIsotope(double mass, double percent);

    public abstract Isotope[] getIsotopes();{

    public abstract double getAvgAtomicMass();
}
```

Now let's consider the *ElementMaker* class below which extends the *Element* class. Notice, in this class, we have implemented the abstract methods above. Also, notice that these methods are proceeded with the keyword *Override*. Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.

Overriding allows us to have multiple implementations of the same method. Having multiple implementations of the same method is referred to as polymorphism. Where "poly" means more than one and "morphism" means many forms.

Element Maker

```
public class ElementMaker extends Element {
    private ArrayList<Isotope> isotopeList = new ArrayList<Isotope>( );

    public ElementMaker(String n, String s, int an){
        super(n, s, an);
    }

    /**
     * Adds an isotope to the isotopeList
     */

    @Override
    public void addIsotope(double mass, double percent) {
       /* Implementation not shown */
    }

    /**
     * Returns the isotopes of an element as an array
     */
    @Override
    public Isotope[] getIsotopes(){
        /* Implementation not shown */
        return isotopes;
    }

    /**
     * Calculates and returns the average atomic mass of an element
     */
    @Override
    public double getAvgAtomicMass() {
        //TODO:  calculate the mass and return the value
        return mass;
    }
}
```

An isotope is a form of the same element that contains equal numbers of protons but different numbers of neutrons in their nuclei, and hence differ in relative atomic mass. Carbon for example has two stable isotopes. Carbon 12 has 6 protons and 6 neutrons and has a relative abundance of 98.9%. Carbon 13 has 6 protons and 7 neutrons and has a relative abundance of 1.1%. Based on this information, the average atomic mass is the weighted average of all the isotopes and can be calculated as follows,

12 * 98.9/100 + 13 * 1.1/100 = 12.011

The *ElementMaker* class shown above creates isotopes using the *Isotope* class below and stores them in an array list. The isotopes associated with a given element can be retrieved using the *getIsotopes* method in the *ElementMaker* class.

```
public class Isotope {
    private double mass;
    private double percent;
    public Isotope(double mass, double percent){
        this.mass = mass;
        this.percent = percent;
    }

    public double getIsotopeMass(){
        return mass;
```

```
    }

    public double getIsotopePercent(){
        return percent;
    }
}
```

Refer to the *ElementMaker* and *Isotope* classes above, then complete the *getAvgAtomicMass* method which calculates and returns the average atomic mass of an element based on its isotopes.

```
    @Override
    public double getAvgAtomicMass() {
        //TODO:  calculate the mass and return the value









        return mass;
    }
```

Now that the abstract methods are implemented, we can create elements and isotopes without understanding how this occurs in the program. For example, in another class called *PeriodicTable* we can create the element hydrogen and its isotopes as shown below. The only piece of information that is needed are the stubs which were defined in the abstract class

Stubs from abstract class *Element*

```
void addIsotope(double mass, double percent);
double getAtomicMass();
```

Calls from the *PeriodicTable* class

```
Element hydrogen = new ElementMaker("Hydrogen", "H", 1);
hydrogen.addIsotope(1.007825, 99.985);
hydrogen.addIsotope(2.01410177811, 0.0115);
System.out.println(hydrogen.getAvgAtomicMass());
```

Refer to the abstract class *Element* and the *ElementMaker* class above. Write code that could be used to create the element copper (Go here to determine the number atomic number for copper, https://ptable.com/). The mass and abundance of the common isotopes of copper are defined below. Use this information to calculate and print the average atomic mass.

Isotope Mass	Abundance
62.9295989	69.17
64.9277929	30.83

The above example illustrates the purpose of abstraction. Notice that with just knowledge of the parameters and return type of the methods, we could create elements, isotopes, and print the corresponding mass of an element. We can now do this for any element!

□ Differentiate between an abstract class and an interface

Another way to achieve abstraction in Java is with an interface. An interface is a completely abstract class that is used to group related methods with empty bodies. Unlike an abstract class however, interfaces cannot have constructors. Interfaces simply provide a framework for the methods to be defined in the classes that inherit them. Once these methods are defined, they can be implemented with understanding their underlying architecture.

Let's revisit the abstract class Element below,

Element

```
public abstract class Element{

    abstract String getName();
    abstract String getSymbol();
    abstract int getAtomicNumber();
    abstract void addIsotope(double mass, double percent);
    abstract Isotope[] getIsotopes();
    abstract double getAvgAtomicMass();
}
```

Notice that everything in this class is abstract. That is, it is completely abstract. As aforementioned, a completely abstract class is also referred to as an interface. We can therefore re-write the above class as an interface,

Element

```
public interface Element{

    String getName();
    String getSymbol();
    int getAtomicNumber();
    void addIsotope(double mass, double percent);
    Isotope[] getIsotopes();
    double getAvgAtomicMass();
}
```

To implement an interface in a subclass we use the keyword implements,

ElementMaker

```
public class ElementMaker implements Element
{
private String name;
private String symbol;
private int atomicNumber;
private ArrayList<Isotope> isotopeList = new ArrayList<Isotope>( );

    public ElementMaker(String n, String s, int an)
    {
        name = n;
        symbol = s;
```

```
        atomicNumber = an;
    }

    public String getName()
    {
        return name;
    }

    public String getSymbol(){
        return symbol;
    }

    public int getAtomicNumber()
    {
        return atomicNumber;
    }

    @Override
    public void addIsotope(double mass, double percent) {
    /* Implementation not shown */
    }

    @Override
    public Isotope[] getIsotopes(){
    /* Implementation not shown */
    }

    @Override
    public double getAtomicMass() {
     /* Implementation not shown */
    }
}
```

Now we can create our elements just as we did before,

```
Element sodium = new ElementMaker("Sodium", "Na", 11);
```

The interface below models an element and its isotopes,

```
public interface Element{

    String getName();
    String getSymbol();
    int getAtomicNumber();
    void addIsotope(double mass, double percent);
    Isotope[] getIsotopes();
    double getAvgAtomicMass();
    void setState(String s);
    String getState(String s);
    void setElectrognegativity(double e);
    double getElectronegativity();
    void setType(String t);
```

```
    String getType();
    Isotope getMostAbundantIsotope();
}
```

The *ElementMaker* class implements the *Element* interface. Elements can be created using the *ElementMaker* class with the constructor shown,

```
    public ElementMaker(String n, String s, int an)
    {
        name = n;
        symbol = s;
        atomicNumber = an;
    }
```

Assume that all the methods in the interface are properly defined in the *ElementMaker* class. Write code to do the following,

(a) Create a new element argon. Go here to determine the argon's symbol and atomic number - https://ptable.com/
(b) Add all the common isotopes associated with argon. Go here to find the masses and abundances of each isotope - https://www.webelements.com/argon/isotopes.html
(c) Set the state of argon to a "gas", "liquid", or "solid"
(d) Set argon's type to "metal", "nonmetal", or "metalloid"
(e) Locate argons's electronegativity. It can be found here - https://ptable.com/

Write a *toString* method that summarizes all the information. Use the methods defined in the interface to get the needed information.

```
@Override
public String toString()
{









}
```

□ Receive credit for this lab guide

Submit this portion of the lab to Pluska to receive credit for the lab guide. Once received, your completed code challenges will also be graded and will count towards your final lab grade.