CSM Week 2 Review

Abstract Classes
The purpose of an abstract class is to have a class that cannot be instantiated. An abstract class can have abstract methods, but do not necessarily need them to be defined as an abstract class.

This is useful because the abstract class is sometimes too vague to be instantiated by itself. For example, consider the abstract class `Shape` below. How do we know the sides and dimension of a shape? Well, we all know that shapes are two-dimensional. However, for sides, it really depends on what shape we are instantiating! Therefore, it would make more sense to not instantiate `Shape`, but instead have it be a class that you can inherit methods from.

```
abstract class Shape {

    //an abstract method that returns the number of sides
    public abstract int sides();

    //This method is commented out because it would yield a compile-time error. Abstract methods cannot be static! However, you can have static concrete methods inside of an abstract class.
    //static abstract int corners():

    //prints out the dimension
    public void dimension() {
        System.out.println("I'm two-dimensional!");
    }

}

public class Square extends Shape {
    public int sides() {
        return 4;
    }
}

public class Triangle extends Shape {
```

```
    public int sides() {
        return 3;
    }
}
```

Here are some examples of how methods would be run:

```
Shape whatShape = new Shape(); //Compile-time Error -> Shape is an abstract class!
Square box = new Square();
Triangle tri = new Triangle();
box.dimension(); //inherits the dimension method from Shape!
box.sides(); //4
tri.sides(); //3
```

Interfaces
Interfaces are useful if you have many unrelated classes that want to implement the same methods in different ways. For example, `Square` and `Planet` from the code block below are competely unrelated classes, but they both want to be able to rotate.

```
interface Rotatable {
    //fields are implicitly static or final
    final boolean canRotate = true;

     //a default method can have an implementation
    public default void spin() {
        System.out.println("I'm spinning!");
    }

    //methods that would be implemented by concrete classes
    public void rotateLeft();
    public void rotateRight();

}
```

```
public class Square implements Rotatable{
    public void rotateLeft() {
        System.out.println("CounterClockwise!");
    }

    public void rotateRight() {
        System.out.println("Clockwise!");
    }
}

public class Planet implements Rotatable {
    public void rotateLeft() {
        System.out.println("I'm not Venus or Uranus!");
    }

    public void rotateRight() {
        System.out.println("I'm either Venus or Uranus!");
    }
}
```

Miscellanious Interface Facts

- Interfaces can **extend**, but cannot **implement** other interfaces.
  - If a concrete class implements an interface that extends another interface, the class must implement all methods in both of the interfaces.
- Interfaces can be used as a static type when creating objects (reference to 1c)
  - Why is this nice? This way, `rotatableObj` can be instantiated as a `Square` or `Planet` object and pass the compiler. Otherwise, we must create a superclass that both `Square` and `Planet` can inherit from, which makes no sense since `Square` and `Planet` are not related.

```
//The code below will pass the compiler!
Rotatable rotatableObj;

rotatableObj = new Square();
```

```
rotatableObj = new Planet();
```

Abstract Classes default Interfaces
Similarities:

- Blueprints for a class
- Both can have static methods
  - Abstract classes can simply have static methods
  - Interfaces can have static methods since Java 8
    - Static methods cannot be overriden by methods in concrete classes
    - Concrete classes that implement the interface cannot use the static method
- Both can have concrete methods
  - Abstract classes can simply have concrete methods
  - Interfaces use the **default** method since Java 8
    - Note: default methods can be overriden in concrete classes

Differences:

- **Usage:** Abstract classes have a "is-a" relationship with other classes and are usually used as a broader descriptor of concrete classes. Interfaces are used to define what a class can do. It doesn't matter what type of class it is. It's actually common for interfaces to be named with the -able suffix.
  - Consider using abstract classes when:
    - You want to share code among several closely related classes.
    - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
    - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
  - Consider using interfaces when:
    - You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
    - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
    - You want to take advantage of multiple inheritance of type.
- **Fields:** Interface fields must be static or final. Abstract class fields are not limited in this sense.
- **Extending/Implementing:** A concrete class can only extend one abstract class, but can implement multiple interfaces.
- **Access control:** Interface methods are implicitly public while abstract class methods are implicitly package private (if you do not change the scope to private, public, or protected). An abstract method cannot be private.

Generics
Generics provide a container that allow you to use any type with the class. In the example below, `T` is a dummy variable that represents a generic type. Note that you could have used any variable name that is not the name of an existing data structure such as `K`, `X`, `Y` to represent a generic type. However, it is convention to name generic types as single uppercase letters.

```
public class Container<T> {

    private T item;
```

```
    public void putItem(T item) {
        self.item = item;
    }

    public T getItem() {
        return item;
    }
}
```

Below are a few examples of how Container would work:

```
Container<String> c = new Container<String>(); //T will now refer to a String
object
c.putItem("cat");
c.putItem(new Point(3, 4)); //Compile-Time Error

Container<Point> p = new Container<Point>(); //T will now refer to a Point
object
c.putItem(new Point(3, 4));

Container<T> t = new Container<String>(); //Compile-Time Error
//T only exists inside the Container class so the symbol T would not exist

Container<Animal> a = new Container<Cat>(); //Cat extends Animal
//While Animal may be a superclass of Cat, Container<Animal> is not a superclass
of Container<Cat>. Therefore, there would be a compile-time error.
```

Iterators
The concept of iterators in CS 61B is very similar to that of iterators in CS 61A. With iterators, you are able to iterate through the elements of a data structure.

Why are iterators interfaces and why are they useful? Well, think about last week when we had the chance to implement a `for each loop` for an integer array.

```java
int[] arr = {1, 2, 3};


//normal for loop
for (int i = 0; i < arr.length; i++) {
    System.out.println(i);
}


//for each loop
for (int i: arr) {
    System.out.println(i);
}


//what the for each loop is actually doing
Iterator arrIterator = arr.iterator();
while (arrIterator.hasNext()) {
    System.out.println(arrIterator.next());
}
```

The normal `for loop` requires more methods regarding the data structure we're iterating through while the `for each loop` implicitly uses an iterator that handles all the specifics "under the hood". An iterator can eliminate an extra layer of abstraction and make your life much more convenient! It also makes sense for `Iterator` to be an interface because many unrelated data structures should have a way of iterating through themselves.

Making a Class Iterable
In the CSM worksheet, we worked on how to create an `Iterator` class for an integer array by implementing the `Iterator` interface. Most existing data structures already implement the `Iterator` interface, so we can simply call their `iterator` method.

However, if we want to use an iterator for a class that we have created ourselves, we would have the class implement the `Iterable` interface.

```java
interface Iterator<T> {
    //tells us if there are more objects to iterate over
    boolean hasNext();


    //tells us the object that will be iterated over
```

```
    T next();
}

interface Iterable<T> {
    Iterator<T> iterator();
}

import java.util.ArrayList;
import java.util.Iterator;

//example of a class implementing Iterable
public class StringList implements Iterable<String> {
    private ArrayList<String> words = new ArrayList<>();

    public void add(String s) {
        words.add(s);
    }

    @Override
    public Iterator<String> iterator() {
        returns new StringListIterator();
    }

    //nested class that implements Iterator
    class StringListIterator implements Iterator<String> {

        private int counter = 0;

        @Override
        boolean hasNext() {
            return counter < words.size();
        }
```

```
        @Override
        String next() {
            String word = words.get(counter);
            counter++;
            return word;
        }
    }
}
```

Resources
Good explanation of interfaces and abstract classes
Abstract Class Documentation
Iterators and Generics