AP CS Unit 7: Interfaces Notes

The purpose of an interface is to specify what an object should be able to do and enforce these behaviors if you implement it. In other words, an interface is a list of public methods without any implementing code. Any class that implements that interface must implement all the methods of the interface. How is up to the programmer!

Let's look at an example. Suppose we have a program with the following classes: Person, Lion, Gold, and Food. All of these objects have something in common: a location defined by x and y locations. Rather than write a superclass, we could write the following interface:

```
import java.awt.Point;

public interface Locatable{
        public Point getLocation();
        public void setLocation( Point p );
        public double getDistance( Locatable thing );
}
```

these must be fleshed out
Headers must match exactly

The idea is that any class that implements this interface must be something that has a location that can be defined by two coordinates and that their location can change.
Point object

Here's what the Lion class might look like

```
import java.awt.Point;

public class Lion implements Locatable{
        private int x, y;
        // other instance variables

        // constructor(s) and other methods

        public Point getLocation(){
                return new Point( x, y );
        }

        public void setLocation( Point p ){
                x = p.x;
                y = p.y;
        }

        public double getDistance( Locatable loc ){
                Point here = getLocation();
                Point there = loc.getLocation();
                return here.distance( there );
        }
}
```

The keyword implements signals that this class must have all the methods of this interface.

The implementing class can have whatever instance variables it needs. It can have any additional methods/constructors it needs.

The method headers must be the same as those specified in the interface though the parameter names may vary.

* public fields are ugly

public class Point
extends Point2D
implements Serializable

To continue with this example, if the Person, Lion, Gold, and Food classes all implement Locatable then we can do things like this:

```
Locatable [] locs = new Locatable[4];
locs[0] = new Lion();
locs[1] = new Person();
locs[2] = new Gold();
locs[3] = new Food();
```

powerful!

You can call any Locatable method without casting. You must cast to call any non-Locatable method.

Or write a method like this:

```
public Point midpoint( Locatable loc1, Locatable loc2 ){
    returns the midpoint between these two objects
}
```

You can pass any objects that implement Locatable to this method.

Note.

- A class may implement multiple interfaces!
- If a superclass implements an interface then the sub class does not need to!
- If a variable's data type is an interface, you may call the interface methods and the methods inherited from the superclass if one exists.

Interfaces, Inheritance, and Data Types

The datatype of a variable can be an abstract class, concrete class, or interface. However, we can only make instances of concrete classes.

ABC can be:
- an abstract class
- an interface
- a concrete class

ABC x = new XYZ();

XYZ must be a concrete class

In other words,

- If ABC is an abstract class, then XYZ must be a concrete class (sub class to ABC)
- If ABC is an interface, then XYZ must be a concrete class implementing ABC
- IF ABC is a concrete class, then XYZ must be a sub class of ABC (any level below)

if this compiles → DEF y = new DEF();

DEF cannot be an abstract class or an interface, it must be a concrete class.

The List Interface.

Reminder!

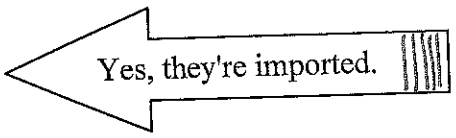
The List interface represents an ordered collection of objects. Each object has a unique position in the list. The List interface has 25 methods. You are responsible for learning 6 of these methods. In the table below, let E refer to the name of a class.

↳ represents generic class.

Method	Description
int size()	returns the logical size of the list.
boolean add(E obj)	appends *obj* to end of list; returns true
void add(int index, E obj)	inserts *obj* at position *index* ($0 \le$ index $\le$ size) moves elements at position *index* and higher to the right (adds 1 to their indices) and adjusts size.
E get(int index)	returns the object located at *index*.
E set(int index, E obj)	replaces the element at position *index* with *obj* returns the element formerly at the specified position
E remove(int index)	removes element from position *index*, moving elements at position index + 1 and higher to the left (subtracts 1 from their indices) and adjusts size. returns the element formerly at the specified position

The ArrayList Class.

The ArrayList class implements the List interface. Internally it stores the data in an array and that's why it's called the ArrayList class. Let's look at an example:




```
import java.util.*;

public class Runner {
        public static void main(String[] args) {
                List<String> list = new ArrayList<String>();
                list.add( "A" );
                list.add( "B" );
                for ( int n = 0; n < list.size(); n++ ){
                        System.out.print( list.get(n) + "," );
                }
        }
}
```

What is displayed?

A, B,

What's up with the < >. Generic Types.

Some classes and interfaces allow you to define the type of objects that the class handles. In the code below, think of List<String> as meaning "a list of strings" and ArrayList<String> as meaning "an array list of strings." Classes and interfaces that support this are called generic types.

```
List<String> list = new ArrayList<String>();
List<Integer> nums = new ArrayList<Integer>();
```

yellow ~~bike~~ red green

1. What is displayed?

yellow

green

```
import java.util.*;

public class Runner{
        public static void main(String[] args) {
                List <String> x = new ArrayList<String>();
                x.add( "red" );
                x.add( "green" );
                x.set( 0, "blue" );
                x.add( 0, "yellow" );
                x.remove( 1 );

                for ( int n = 0; n < x.size(); n++ ) {
                        String s = x.get(n);
                        System.out.println( s );
                }
        }
}
```

2. What is displayed?

7
8
3

```
import java.util.*;

public class Runner{
    public static void main(String[] args) {
        List <Cat> x = new ArrayList<Cat>();
        System.out.println( x.size() );
        x.add( new Cat( 7 ) );
        x.add( 1, new Cat( 3 ) );
        x.add( 1, new Cat( 8 ) );

        for ( Cat c : x )
            System.out.println( c.get() );
    }
}
```

```
public class Cat{
    private int x;

    public Cat( int x){
        this.x = x;
    }

    public int get(){
        return x;
    }
}
```

7 3 8

3. What is displayed? (There is a very tricky part to this).

5 5 6

5 5 6

```
List<Integer> list = new LinkedList<Integer>();
list.add( 5 );
list.add( 5 );
list.add( 6 );

for ( Integer i : list )
        System.out.print( i + " " );
System.out.println();

for ( int k = 0; k < list.size(); k++ ){
        if ( list.get(k) == 5 )
                list.remove(k);
}

for ( Integer i : list )
        System.out.print( i + " " );
System.out.println();
```

5 5 6

skips over show on board trace variables.

5 6