This practice exam will help you prepare for the first midterm. Have some paper ready. For the real midterm, the exam will include extra pages to do work. There is an appendix at the end with methods that may be helpful. This practice midterm is intended to take you 90 minutes as the actual exam will. You will not be allowed any cheatsheet or other extra materials for the exam.

```
/**
 *     This StringTools class contains some useful methods that work with Strings.
 */
public class StringTools {

       /**
        * Return the count of consonants minus the count of vowels (a,e,i,o,u)
        * @param s The string to analyze.  If it is null, an exception will be thrown
        * @return The calculated number.  If the parameter does not contain a string with
characters, returns 0.
        */
       public static int consMinusVowels(String s) {...}

}
```

What test cases should be run to test this method? DO NOT WRITE A FULL JUNIT TEST METHOD OR ASSERTIONS.
Write only the argument values to pass in for the test cases. Write a few words about each test case explaining why it is important.

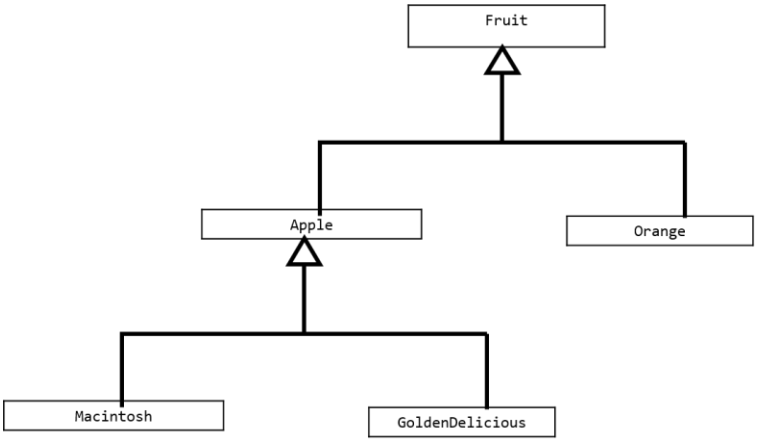
INPUT	EXPLANATION
"bed"	

Create the class specified by the UML diagram below and specifications that follow. It represents a roadtrip itinerary for a traveler. It will contain two data members, the traveler's name and the list of cities they will be in. Each element in citiesByDay represents one day, so a city may appear multiple times in the list.

Itinerary
-travelerName: string -citiesByDay: ArrayList<String>
+Itinerary(String n, ArrayList<String> cities) +Itinerary(String n) +getName(): String +getCitiesByDay(): ArrayList<String> +citiesVisited(): ArrayList<String> +numberOfCitiesVisited(): int +numberOfDays(): int +addNextCity(String city): void +resetItinerary(): void +toString(): String +*daysTogether(Itinerary i1, Itinerary i2): int*

- The class contains two data members
- Any time a parameter value is not usable, throw an exception
- Create two constructors. One sets the name, the other sets the name and provides an arraylist of cities.
- getName, getCitiesByDay return the appropriate data
- citiesVisited returns a list of unique cities visited; the citiesByDay list, but with duplicates removed
- numberOfCitiesVisited returns the number of unique cities visited
- numberOfDays(): returns how many days long the itinerary is
- +addNextCity(String city): adds the city to the end of the citiesByDay list
- +resetItinerary(): empties the citiesByDay list
- +toString(): output the name and citiesByDay list in a reasonable format
- +daysTogether(Itinerary i1, Itinerary i2): A static method that given two itineraries, calculates on how many days both travelers will be in the same city. {"Vancouver", "Vancouver", "Victory"} and {"Nanaimo", "Vancouver", "Vancouver", "Victoria"} would be 1 because they only match on the second day.

1. Given the following diagram, answer the following questions



Given the declaration:

```
Fruit fruit = new GoldenDelicious();
Orange orange = new Orange();
```

a. Is fruit instanceof Fruit true?
b. Is fruit instanceof Orange true?
c. Is fruit instanceof Apple true?
d. Is fruit instanceof GoldenDelicious true?
e. Is fruit instanceof Macintosh true?
f. Is orange instanceof Orange true?
g. Is orange instance of Fruit true?
h. Is orange instance of Apple true?
i. Suppose Apple contains a method called makeApple(). Can fruit invoke makeApple()? Cane orange invoke makeApple()?
j. Suppose makeOrangeJuice() is defined in Orange. Can orange invoke this method? Cand fruit invoke this method?
k. Is the statement Orange p = new Apple() legal?
l. Is the statement Macintosh p = new Apple() legal?
m. Is the statement Apple p = new Macintosh() legal?

There are errors in the following code. Correct them:

```
ArrayList list = new ArrayList();
list.add(“Burnaby”);
list.add(“Vancouver”);
list.add(new java.util.Date());
String city = list.get(0);
list.set(3, “Richmond”);
System.out.println(list.get(3));
```

```
public abstract class Animal {
	public String action() {
		return "lives";

	}

	public String toString() {
		return "This animal " + this.action();
	}
}


public class Bird extends Animal {
	private boolean flies;

	public Bird(boolean canFly) {
		super();
		this.flies = canFly;
	}

	public boolean canFly() {
		return this.flies;
	}

	public String action() {
		String result = super.action();
		if (this.flies) {
			result = result + " and flies";
		}
		return result;
	}
}


public class Parrot extends Bird {
	public Parrot() {
		super(true);
	}

	public String action() {
		return super.action() + " and talks";
	}
}


public class Coral extends Animal {
	public Coral() {
		super();
	}

	public String toString() {
		return "This can't be an amimal";
	}
}
```

For each of the following pieces of code, either write what will be displayed to the console **or** if the code would create an error, **clearly** describe the cause of the error. [2 marks each]

```
Animal v = new Animal();
System.out.println(v);
```

```
Animal s = new Bird(true);
System.out.println(s.canFly());
```

```
Parrot p = new Bird(true);
System.out.println(p.action());
```

```
Animal c = new Coral();
System.out.println(c);
```

```
Bird m = new Parrot();
System.out.println(m);
```

```
Parrot x = new Parrot();
System.out.println(x.canFly());
```

```
Coral k = new Coral();
System.out.println(k.action());
```

```
Animal d = new Bird(false);
System.out.println(d);
```

```
Coral j = new Coral();
System.out.println(j);
```

Appendix

An abbreviated list of the methods of different classes we have used in this course.

The ArrayList class

Constructors	
	ArrayLIst<E>() Creates an empty ArrayList with initial capacity of ten.
	ArrayLIst<E>(A**rrayList<E>** a) Creates a copy of the parameter ArrayList.
Modifier and Type	**Method and Description**
boolean	**add**(**E** e) Appends the specified element to the end of this list.
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list.
boolean	**contains**(**Object** o) Returns true if this list contains the specified element.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(**Object** o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
int	**lastIndexOf**(**Object** o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(**Object** o) Removes the first occurrence of the specified element from this list, if it is present.
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
String	**toString()** Returns a string representation of this collection. The string representation consists of the collection's elements in order, enclosed in square brackets () "[]". Adjacent elements are separated by the characters ", "

The String Class

`char`	`charAt(int index)` Returns the character at the specified index.
`int`	`compareTo(String anotherString)` Compares two strings lexicographically.
`String`	`concat(String str)` Concatenates the specified string to the end of this string.
`boolean`	`equals(String str)` Compares this String to another string.
`boolean`	`equalsIgnoreCase(String str)` Compares this String to another string ignoring case considerations.
`int`	`indexOf(int ch)` Returns the index within this string of the first occurrence of the specified character.
`int`	`indexOf(int ch, int fromIndex)` Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
`int`	`indexOf(String str)` Returns the index within this string of the first occurrence of the specified substring.
`int`	`indexOf(String str, int fromIndex)` Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
`int`	`lastIndexOf(int ch)` Returns the index within this string of the last occurrence of the specified character.
`int`	`lastIndexOf(int ch, int fromIndex)` Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
`int`	`lastIndexOf(String str)` Returns the index within this string of the rightmost occurrence of the specified substring.
`int`	`lastIndexOf(String str, int fromIndex)` Returns the index within this string of the last occurrence of the specified substring.
`int`	`length()` Returns the length of this string.
`String`	`replace(char oldChar, char newChar)` Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
`String`	`substring(int beginIndex)` Returns a new string that is a substring of this string.
`String`	`substring(int beginIndex, int endIndex)` Returns a new string that is a substring of this string.
`String`	`toLowerCase()` Converts all of the characters in this `String` to lower case using the rules of the default locale, which is returned by `Locale.getDefault`.
`String`	`toUpperCase()` Converts all of the characters in this `String` to upper case using the rules of the default locale, which is returned by `Locale.getDefault`.