1 ADT Selection

Suppose we have a TA Shreyas who teaches multiple discussion sections! A student may frequent more than one discussion section. For each situation below, choose the best ADT(s) out of the following — Map, Set, List — and explain how you can use the ADT(s) to solve the problem. Each subpart is independent of the previous. One answer may involve multiple ADTs. There may be multiple efficient answers for each problem.

1. Storing all the Students in Shreyas’s first section in alphabetical order.

2. Storing all the Students by their section, where Students within a section are sorted alphabetically.

3. Storing the Students in all of Shreyas’s sections. There shouldn’t be duplicates.

4. Quickly getting a Student by sid.

5. Quickly getting all Students of a given name. Names aren’t necessarily unique.

6. Cycling through the Students in one discussion section.

Solution:

1. Put the Students in a List in alphabetical order.

2. Use a Map, where each Section maps to an alphabetically ordered List of Students in that section.

3. Use a Set. Add all the Students to the Set. Since a set requires elements to be unique, calling add on a student already in the set will not add a duplicate.

4. Use a Map, where each sid maps to one Student.
5. Use one Map that maps names to a List (or Set) of Students of the given name.

6. Put the Students in a List. You could use a LinkedList and repeatedly remove from the front and reinsert at the back. Equivalently, you could use an ArrayList and keep an index pointer.
2 The ABCs of OOP

Indicate what each line the main program in class D would print, if the line prints anything. If any lines error out, identify the errors as compile time or runtime errors.

```java
public class A {
    public void x() {
        System.out.println("Ax");
    }

    public void y(A z) {
        System.out.println("Ay");
    }
}
public class B extends A {
    public void y() {
        System.out.println("By");
    }

    public void y(B z) {
        System.out.println("Byz");
    }
}
public class C extends A {
    public void x() {
        System.out.println("Cx");
    }
}
public class D {
    public static void main(String[] args) {
        A e = new B();
        A f = new C();
        B g = new A();
        B h = new C();
        C i = (C) new A();
        B j = (A) new C();
        B k = (B) e;
        f.x();
        e.x();
        e.y();
        ((B) e).y();
        e.y(e);
        e.y(f);
    }
}
```
A e = new B();
A f = new C();
B g = new A(); // compile time error
B h = new C(); // compile time error
C i = (C) new A(); // runtime error
B j = (A) new C(); // compile time error
B k = (B) e;
f.x(); // Cx
e.x(); // Ax
e.y(); // compile time error
((B) e).y(); // By
e.y(e); // Ay
e.y(f); // Ay
3 Classy Cats

Look at the `Animal` class defined below. The `protected` access modifier may be new to you. Simply put, it gives classes in the same package and subclasses access to those variables. Don’t worry too much about understanding this - it’s not in scope for exams.

```java
public class Animal {
    protected String name, noise;
    protected int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
        this.noise = "Huh?";
    }

    public String makeNoise() {
        if (age < 2) {
            return noise.toUpperCase();
        }
        return noise;
    }

    public String greet() {
        return name + ": " + makeNoise();
    }
}
```

(a) Given the `Animal` class, fill in the definition of the `Cat` class so that it makes a "Meow!" noise. Assume this noise is all caps for kittens, i.e. `Cats` that are less than 2 years old.

```java
public class Cat extends Animal {
    public Cat(String name, int age) {
        super(name, age);
        this.noise = "Meow!";
    }
}
```

Solution:
Explanation: Inheritance is powerful because it allows us to reuse code for related classes. With the Cat class here, we just have to re-write the constructor to get all the goodness of the Animal class. Why is it necessary to call super(name, age); within the Cat constructor? It turns out that a subclass’s constructor by default always calls its parent class’s constructor (aka a super constructor). If we didn’t specify the call to the Animal super constructor that takes in a String and a int, we’d get a compiler error. This is because the default super constructor (super();) would have been called. Only problem is that the Animal class has no such zero-argument constructor! By explicitly calling super(name, age); in the first line of the Cat constructor, we avoid calling the default super constructor.

Similarly, not providing any explicit constructor at all in the Cat implementation would also result in code that does not compile. This is because when there are no constructors available in a class, Java automatically inserts a no-argument constructor for you. In that no-argument constructor, Java will then attempt to call the default super constructor, which again, does not exist.

(b) "Animal" is an extremely broad classification, so it doesn’t really make sense to have it be a class. Look at the new definition of the Animal class below.

```java
public abstract class Animal {
    protected String name;
    protected String noise = "Huh?";
    protected int age;

    public String makeNoise() {
        if (age < 2) {
            return noise.toUpperCase();
        }
        return noise;
    }

    public String greet() {
        return name + ": " + makeNoise();
    }

    public abstract void shout();
}
```

Fill out the Cat class again below to allow it to be compatible with Animal (which is now an abstract class) and its one methods.

```java
public class Cat extends Animal {
    public Cat() {
        this.name = "Kitty";
    }
}
```
```java
    this.age = 1;
    this.noise = "Meow!";
}

public Cat(String name, int age) {
    this();
    this.name = name;
    this.age = age;
}

@Override
    public void shout() {
        System.out.println(noise.toUpperCase());
    }
```

**Solution:**

```java
public class Cat extends Animal {
    public Cat() {
        this.name = "Kitty";
        this.age = 1;
        this.noise = "Meow!";
    }

    public Cat(String name, int age) {
        this();
        this.name = name;
        this.age = age;
    }

    @Override
        public void shout() {
            System.out.println(noise.toUpperCase());
        }
}
```

**Explanation:** To override an abstract method, the method signature’s access modifiers must match exactly. Since shout is declared to be public abstract in Animal, our Cat class must declare it to be public to ensure that access modifiers match. Note the Override tags aren’t required, but are good practice.