UNIVERSITY AT ALBANY
State University of New York

COLLEGE OF ENGINEERING AND APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

ICSI213/IECE213 Data Structures

Project 02 by Qi Wang

The following parts are included in this document:

- Part I: General project information
- Part II: Project grading rubric
- Part III: Examples on complete a project from start to finish
- Part IV: A. How to test?
  B. Project description

Proper use of the course materials including the source codes:

All course materials including source codes/diagrams, lecture notes, etc., are for your reference only. Any misuse of the materials is prohibited. For example,

- Copy the source codes/diagrams and modify them into the projects.
  - Students are required to submit the **original work** for the projects. **For each project, every single statement for each source file and every single class diagram for each design (if any) must be created by the students from scratch.**
- Post the source codes, the diagrams, and other materials online.
- Others

Part I: General Project Information

- All work is individual work unless it is notified otherwise.
- Work will be rejected with no credit if
  - The work is not submitted via Duifene.
  - The work is late.
  - The work is partially or entirely written in Chinese.
  - The work is not submitted properly.
    - Blurry, wrong files, not in required format, crashed files, etc.
  - The work is a copy or partial copy of others' work (such as work from another person or the Internet).
- Students must turn in their original work. Any cheating violation will be reported to the college. Students can help others by sharing ideas, but not by allowing others to copy their work.
- Documents to be included/submitted as a zipped folder:
  - UML class diagram(s) – created with Violet UML or StarUML
  - Java source file(s) with Javadoc style inline comments
  - Supporting files if any (For example, files containing all testing data.)

  Lack of any of the required items or programs with errors will result in a low credit or no credit.

- **How to prepare a zipped folder for work submission?**
  - Copy the above-mentioned files into a folder, rename the folder using the following convention:
    [*Your_first_name*][*your_last_name*]*ProjectNumber*
    - For example*, JohnSmithProject02*
  - Zip the folder. A zipped file will be created.
    - For example*,* a file with name *JohnSmithProject01.zip* will be created*.*
  - Submit the zip file on Duifene.
  - You must submit a project in this format. **Submissions not in the required format may be rejected or will result in a low credit or no credit.**

- **Grades and feedback**: Co-instructors will grade. Feedback and grades for properly submitted work will be posted on Duifene. For questions regarding the feedback or the grade, students should reach out to their co-instructors first. Students have limited time/days from when a grade is posted to dispute the grade. Check email daily for the grade review notifications sent from the co-instructors. **Any grade dispute request after the dispute period will not be considered.**

Part II: Project grading rubric

Project 2: (100 points)	**LEVELS OF PERFORMANCE INDICATORS**				**Points Earned**
Performance Indicator (PI)	**UNSATISFACTORY**	**DEVELOPING**	**SATISFACTORY**	**EXEMPLARY**	
PI1: (10 points) UML design	None/Not correct at all. (0)	Visibility, name, and type/parameter type/return type present only for some members for some classes. Some class relationships are incorrect. (<=5)	Visibility, name, and type/parameter type/return type present for each member for all classes with minor issues, class relationships are correct with minor issues. (<=7)	Visibility, name, and type/parameter type/return type present for each member for all classes without issues, class relationships are correct. (<=10)	
	Instructor's Comments:				
PI2(5 points): Comments (Javadoc format) All tags must be included correctly.	None (0)	Some comments are written properly. (<=2)	Most comments are written properly. (<=4)	All comments are written properly. (<=5)	
	Instructor's Comments:				
PI3(5 points): Variable/method naming	Single-letter names everywhere	Many abbreviations (<=2)	Full words most of the time (<=4)	Full words, descriptive (<=5)	
	Instructor's Comments:				
PI4(20 points): The interface	No methods are designed correctly. (0)	Some of the methods are designed correctly. Have a lot of issues. (<=5)	Most of the required methods are designed correctly. Have minor issues. (<=15)	All required methods are designed correctly. Have no issues. (<=20)	
	Instructor's Comments:				
PI5(30 points): The reference-based implementation of the interface *A doubly linked list*	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=10)	Most of the required methods are implemented correctly. Have minor issues. (<=20)	All required methods are implemented correctly. Have no issues. (<=30)	
	Instructor's Comments:				
PI6(20 points):	No methods are tested correctly.	Some of the methods are tested correctly.	Most of the required methods are tested	All required methods are tested	

The driver program and its testing helper class	(0)	Have a lot of issues. (<=5)	correctly. Have minor issues. (<=15)	correctly. Have no issues. (<=20)	
	Instructor’s Comments:				
PI7(10 points): A *generic* Node class	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=5)	Most of the required methods are implemented correctly. Have minor issues. (<=7)	All required methods are implemented correctly. Have no issues. (<=10)	
	Instructor’s Comments:				
				Not submitted as a zipped folder	-10
				Total out of 100	
				Total out of 60	

Part III: Examples on complete a project from start to finish

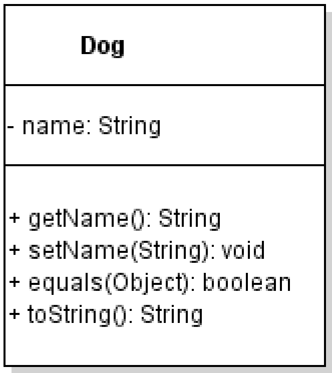
To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

Analysis-design-code-test/debug-documentation.

1) Read project description to understand all specifications (**Analysis**).
2) Create a design (an algorithm for method or a UML class diagram for a class) (**Design**)
3) Create Java programs that are translations of the design. (**Code/Implementation**)
4) Test and debug, and (**test/debug)**
5) Complete all required documentation. (**Documentation**)

The following shows a sample design and the conventions.

- *Constructors* and *constants* should not be included in a class diagram.
- For each *field* (instance variable), include *visibility*, *name*, and *type* in the design.
- For each *method*, include *visibility*, *name*, *parameter type(s)* and *return type* in the design.
    - DON'T include *parameter names*, only *parameter types* are needed.
- Show class relationships such as dependency, inheritance, aggregation, etc. in the design. Don't include the driver program or any other testing classes since they are for testing purpose only.
    - Aggregation: For example, if Class A has an instance variable of type Class B, then, A is an aggregate of B.



The corresponding source codes with inline Javadoc comments are included on next page.

```
import java.util.Random;

/**
 * Representing a dog with a name.
 * @author Qi Wang
 * @version 1.0
 */
public class Dog{
        /**
         * The name of this dog
         */
        private String name;

        /**
         * Constructs a newly created Dog object that represents a dog with an empty name.
         */
        public Dog(){
                this("");
        }
        /**
         * Constructs a newly created Dog object wit
         * @param name The name of this dog
         */
        public Dog(String name){
                this.name = name;
        }

        /**
         * Returns the name of this dog.
         * @return The name of this dog
         */
        public String getName(){
                return this.name;
        }

        /**
         * Changes the name of this dog.
         * @param name The name of this dog
         */
        public void setName(String name){
                this.name = name;
        }

        /**
         * Returns a string representation of this dog. The returned string contains the type of
         * this dog and the name of this dog.
         * @return A string representation of this dog
         */
        public String toString(){
                return this.getClass().getSimpleName() + ": " + this.name;
        }

        /**
         * Indicates if this dog is "equal to" some other object. If the other object is a dog,
         * this dog is equal to the other dog if they have the same names. If the other object is
         * not a dog, this dog is not equal to the other object.
         * @param obj A reference to some other object
         * @return A boolean value specifying if this dog is equal to some other object
         */
        public boolean equals(Object obj){
                //The specific object isn't a dog.
                if(!(obj instanceof Dog)){
                        return false;
                }
                //The specific object is a dog.
                Dog other = (Dog)obj;
                return this.name.equalsIgnoreCase(other.name);
        }
}
```

TAB

TAB

TAB

TAB

open {

open {

Class comments must be written in Javadoc format before the class header. A **description** of the class, author information, and version information are required.

Comments for fields are required.

Method comments must be written in Javadoc format before the method header. The first word must be a verb in title case and in the **third** person. Use punctuation marks properly.

A description of the method, comments on parameters if any, and comments on the return type if any are required.

A Javadoc comment for a formal parameter consists of three parts:

- parameter tag,
- a name of the formal parameter in the design ,
  (The name must be consistent in the comments and the header.)
- and a phrase explaining what this parameter specifies.

A Javadoc comment for return type consists of two parts:

- return tag,
- and a phrase explaining what this returned value specifies

More inline comments can be included in single line or block comments format in a method.

Part IV:

A. How to test?

There can be many classes in a software design. **A UML class diagram should contain the designs of all classes and the class relationships (For example, is-a, dependency, or aggregation).**

- First, test each class separately in its own driver.
  - Create instances of the class (If a class is abstract, the members of the class will be tested in its subclasses.). For example, the following creates Dog objects.

    Create a default Dog object.

    ```
    Dog firstDog = new Dog();
    ```

    Create a Dog object with a specific name.

    ```
    Dog secondDog = new Dog("Sky");
    ```

  - Use object references to invoke instance methods. If an instance method is a value-returning method, call this method where the returned value can be used. For example, method getName can be called to return firstDog's name. You may print the value stored in firstDogName to verify.

    ```
    String firstDogName;
      …
    firstDogName = firstDog.getName();
    ```

  - If a method is a void method, invoke the method to simply performs the task. Use other method to verify that the method performs the task properly. For example, setName is a void method that changes the name of this dog. After this statement, the secondDog's name is changed to "Blue".

    ```
    secondDog.setName("Blue");
    ```

    Print the return value from getName to verify that the name has been changed correctly.

  - Repeat until all methods are tested.

- Next, test the entire design by creating a driver program and a helper class for the driver program. When completing these two classes, you will test and learn how to use the classes (e.g., Dog class) that you have designed. You may choose different identifiers for the class names. For this project, Driver and Helper are used as the class identifiers.
  - Create a helper class for the driver. The helper class can begin with the following three static methods.

```
public class Helper{
   public static void start(){
           This void method is decomposed into a few tasks:
           Create an empty list.
           Fill the list by calling create with a reference to the list.
           Print the list by calling display with a reference to the list.
          …
   }

   public static returnTypeOrVoid create(a reference to a list) {
           Create objects using data from an input file and store the objects into the list.
            …
   }

   public static returnTypeOrVoid display(a reference to a list) {
           Display the objects of the list.
          …
   }
   …
}
```

- Create a driver program. In *main* of the driver program, call *start* to start the entire testing process.

```
public class Driver{
   public static void main(String[] args){
         Helper.start();
   }
}
```

- There are more methods than what are invoked/tested by the previous three methods. You may add more statements in the methods or write additional methods to test the rest of the methods for all classes.
- **T**he driver and its helper class are for testing purpose only. They should not be included in the design diagram. But you will need to submit their source codes.

B. Project description

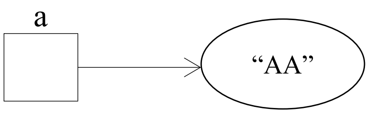
ADT *LinkedString*

Java *String* class is composed of a collection of characters and a set of operations on the characters, an ADT. For this project, you will create a reference-based implementation (called *LinkedString*) on some *String* operations such as *chartAt, concat, isEmpty, length, substring*. A doubly linked list must be used as the data structure. All objects of *LinkedString* must be immutable objects.

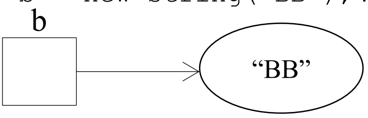
In Java, a *string* is an immutable object (its internal states cannot be changed once it's created). An immutable object(string) can't be altered once it is made by a constructor and the constructor has completed execution. This is useful as you can pass references to the object around, without worrying that someone else is going to change its contents. Any method that is invoked which seems to modify the value, will create another String. For example, three *String* objects, `a`, `b`, and `ab`, can be made after the following code segment is executed.

```
String a = new String(“AA”);
String b = new String(“BB”);
String ab = a.concat(b);
```

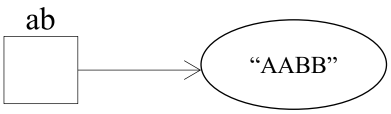
After `String a = new String(“AA”);` is executed, a new *String* `a` is created.



After `String b = new String(“BB”);` is executed, another new *String* `b` is created.



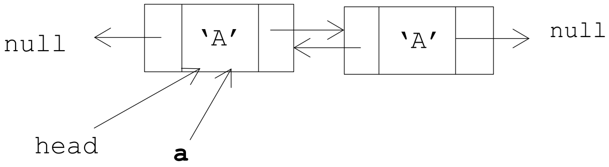
After `String ab = a.concat(b);` is executed, another new *String* `ab` is created. *String* `a` (*this* string) and *String* `b` (a string passed into method `concat`) are not changed. Method `concat` simply makes a new String object with the contents of `a` and `b` copied into the new string object.



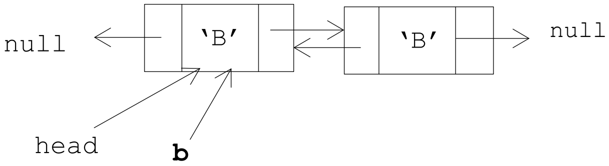
The *LinkedString* class must be implemented so that *LinekdString* objects are immutable. The *LinkedString* class must use a doubly linked list, a different data structure from the one used by the *String* class, to store a collection of characters. Characters are stored in their own nodes, each of which has a reference to the previous node and a reference to the next node. This data structure(the doubly linked list) is *LinkedString* 's internal state. An immutable *LinkedString* object means its linked list can't be altered once the object is created. All characteristics and behaviors of *LinkedString* class must be designed with the same logic as Java *String* class. When a *LinkedString* object calls a method, this *LinkedString* object and the *LinkedString* object(s) passed into this method must be unchanged during execution of this method. If the method returns a *LinkedString* object, a new *LinkedString* object must be made. The following shows how object immutability can be enforced when implementing method `concat`. Three *LinkedString* objects, `a`, `b`, and `ab`, can be made after the following code segment is executed.

```
LinkedString a = new LinkedString ("AA");
LinkedString b = new LinkedString ("BB");
LinkedString ab = a.concat(b);
```

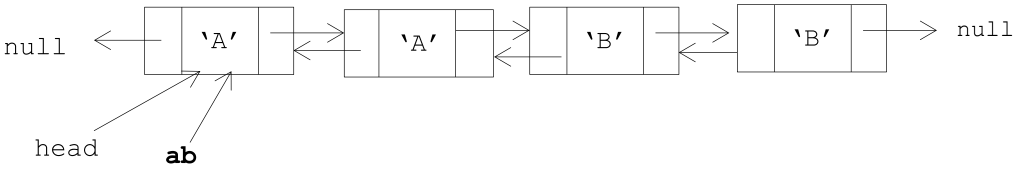
After `LinkedString a = new LinkedString("AA");` is executed, a new *LinkedString* object a is created with all characters stored in a doubly linked list. Each node contains a *Character* element, a successor, and a predecessor.



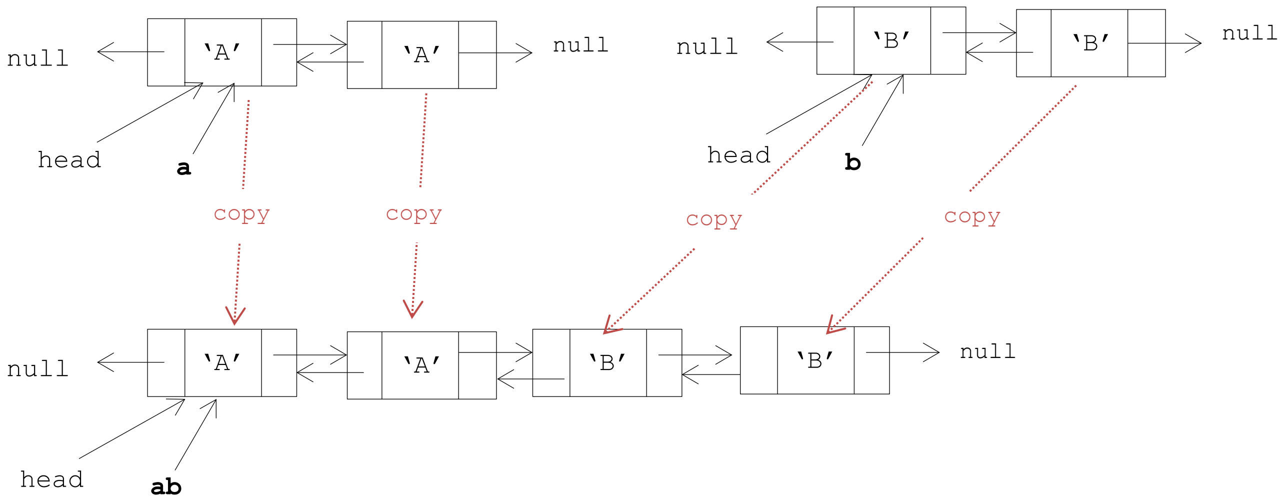
After `LinkedString b = new LinkedString("BB");` is executed, another new *LinkedString* object `b` is created with all characters stored in a doubly linked list. Each node contains a *Character* element, a successor, and a predecessor.



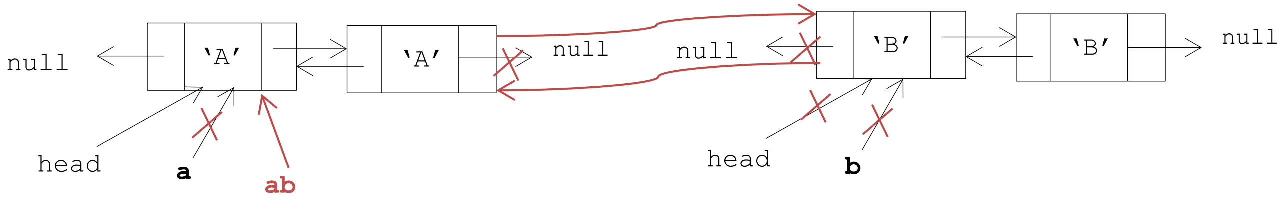
After `LinkedString ab = a.concat(b);` is executed, another new *LinkedString* object `ab` is created with all characters copied/stored in a doubly linked list. Each node contains a *Character* element, a successor, and a predecessor.



Method `concat` must be implemented in a way in which a new linked string is made without modifying this linked string `a` and the other linked string `b` to enforce object immutability. To do this, method `concat` should simply copy characters from `a` and `b` and use them to make a new linked string.



Modifying this linked string `a` or other linked string `b` as follows would violate *LinkedString* object immutability property.



Carefully implement each method, and make sure object immutability property is maintained.

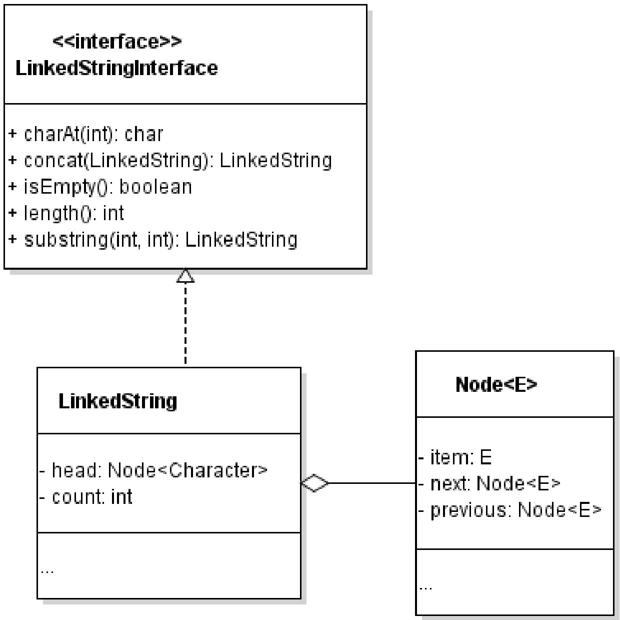
Specification/Analysis:

- **A Generic Node:**
  Create a generic *Node<**E**>* that can be instantiated to create a node containing an element of **any reference type**, a successor, and a predecessor. When using *Node<E>* in *LinkedString*, replace the type parameter with reference type *Character*.
- ***LinkedString* Operations:**
  Note: All operations must be implemented to enforce object immutability. This means this linked string and other linked strings involved cannot be modified once after they are created.
  ✓ return the *char* value at the specified index. (*char charAt(int)*).
  ✓ concatenate a specified linked string to the end of this linked string (*LinkedString concat(LinkedString)*).
  ✓ returns true if and only if the length of this linked string is 0. (*boolean isEmpty()*)
  ✓ return the length of this linked string (*int length()*).
  ✓ return a new linked string that is a substring of this linked string (*LinkedString substring(int, int)*).
  To find out more information regarding the exceptions of some of the above-mentioned methods, go to the *String* class in the Java library. You don't need to create user-defined exceptions but using the java exceptions that are already included in the designs of these methods.

Design:

Complete a UML diagram to include all classes. An interface class is usually defined to specify what the operations do. A class implementing this interface provides implementations to specify how the operations are implemented.

In the design, you should include the design of *LinkedString*, the design of a *Node* class, etc. The following shows **partial** design for this project.



Code/Implementation:

Implementation includes selecting a data structure, implementation of constructors and other operations.

A doubly linked list with an external reference to the *head* must be used as the data structure. A doubly linked list is a reference to the head of a doubly linked list. An instance variable *count* counts the number of *LinkedList* objects that have been created.

Three overloading constructors should be provided to make a linked string from an empty list, a *char* array, or a *String* object.

- create an empty *LinkedString* instance.

  A new character linked list is created and it contains 0 characters (*LinkedString()*).

- create a *LinkedString* instance from a sequence of characters.

  A new character linked list is created. It contains the same sequence of characters as the *char[]* argument and each of character is stored in its own node (*LinkedString(char[]*)).

- create a *LinkedString* instance from a *String* object.

  A new character linked list is created. It contains the same sequence of characters as the *String* argument and each character is stored in its own node (*LinkedString(String)*).

When implementing overloading methods/constructors, you should write all the codes in the constructor that has most parameters and let other constructors to invoke/reuse this constructor. In this case, the last two constructors all have one parameter. Because a *String* object can be converted into a *char* array, therefore, all the codes should be written in the second constructor. The first constructor and the third constructor should invoke the second constructor using *this* reference.

When implementing other methods, make sure all *LinkedString* objects involved are immutable. It may be helpful to create helper methods for some of the *LinkedString* methods. If so, those helper methods should be private methods. Javadoc comments should also be included. Class comments must be included right above the corresponding class header. Method comments must be included right above the corresponding method header. All comments must be written in Javadoc style.

Debug/Testing:

Note: It is required to store all testing data in an input text file.

It is not efficient to write everything in *main*. Method *main* should be small and the only method in a driver program. A helper class for the driver can begin with the following three methods.

- Create a helper class of the driver program with following methods included.

```
public class Helper{
        public static void start(){
                This void method is decomposed into a few tasks:
                Create an empty array list.
                Fill the list by calling create with a reference to the list.
                Print the list by calling display with a reference to the list.
                Call otherMethods with a reference to the list for more testing.
          …
        }

        public static returnTypeOrVoid create(a reference to a list) {
              Create objects using data from an input file and store the objects into the list.
                -    While there are more tokens,
                        o    Read the next token from the input file as a string, use it to create a linked string and add it
                             to the array list
                -    Make a few empty linked strings and add them into the array list.
                Note: This will test the constructors.
              …
        }

        public static returnTypeOrVoid display(a reference to a list) {
                Display/print the linked strings from the list.
                Note: This will test toString method.
          …
        }
    …
}
```

- There are more methods than what are invoked/tested by the previous three methods. You may add more statements in the methods or write additional methods to test the rest of the methods for all classes. Can you create another static method that test other operations?

```
        public static returnTypeOrVoid otherMethods(a reference to a list) {
                Given the array list of LinkedString objects, for each linked string,
                        -    print if it is empty (isEmpty).
                        -    print the node at index 0 (charAt).
                        -    Print the length (length).
                        -    Print the substring from 0 to 1 (The end index is exclusive.) (substring).
                        -    Concatenate it with its predecessor in the list and print the concatenated result (concat).
                 Note: toString method must be implemented in both Node and LinkedString classes to print
                           properly.
                  …
        }
```

- Create a driver program. In *main* of the driver program, call method *start* to start the entire testing process.

```
              public class Driver{
                 public static void main(String[] args){
                       Helper.start();
```

```
            }
        }
```

The driver and its helper class are for testing purpose only. They should not be included in the design diagram. But you will need to submit their source codes.

The sample testing file may contain items like this:

```
Olivia
Oliver
Amelia
Harry
Isla
Jack
Emily
George
Ava
Noah
Lily
Charlie
Mia
Jacob
Sophia
Alfie
Isabella
Freddie
Grace
Oscar
```

…

Documentation:

Complete all other documents needed.