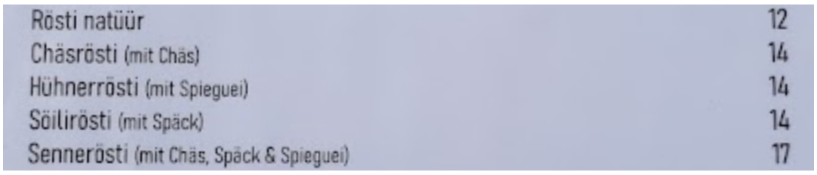
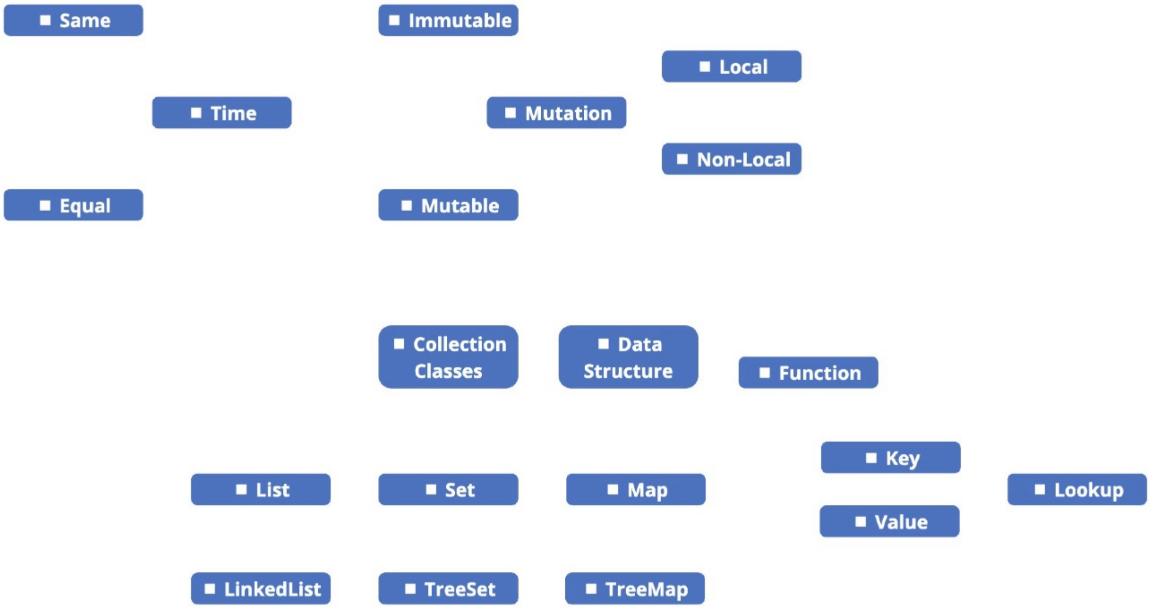
Mutable Lists & Maps

Student name:	TA signature:



Concepts Check off understood concepts, connect related concepts, label connections



Make sure you can **explain** each concept and each connection, you can provide **examples**, and you can **identify** them in a given piece of code.

Names Circle the methods, underline the types

`java.util.List` • `java.util.LinkedList` • `java.util.Map` • `java.util.TreeMap` • `java.util.Set` • `java.util.TreeSet`

Mutable Collections

Our `Sequence` interface and its implementations are immutable. If you want to "modify" a sequence, you have to create a new one. For example, the following `cons` instance method takes a value, and returns a **new** sequence with that value as its first element and the original sequence (`this`) as the rest:

```
public interface Sequence<T> {
  …
  public Sequence<T> cons(T value);
}
```

Thus, "knowing" a sequence means knowing that exact sequence with that exact content. If you modify the content, you get a **new** sequence.

```
public void Sequence<Person> befriend(Sequence<Person> friends, Person f) {
  Sequence<Person> newFriends = friends.cons(f);
  // draw stack and heap state at this point
  assert newFriends != friends;
  return newFriends;
}
```

Draw a memory diagram (stack and heap) if the above method is called like this: `befriend(of(new Person(1), new Person(2)), new Person(9))`

Draw all `Cons`, `Empty`, and `Person` objects. Don't draw stack frames that already got popped. Assume the following Person type: `public record Person(int id) {}`.

In the above method, `friends` and `newFriends` refer to two **different** objects!

With our immutable design, any method that wants to "modify" a sequence needs to **return the modified sequence**. In general, in a pure immutable world, any method that wants to "mutate" something has to create and return a new object.

There is a way around this! But it comes at a cost. Now that we have **mutable** instance variables, we can create methods that don't **need** to return results of "mutation". For example, we can create a class `List` that has a mutable instance variable of type `Sequence`. The `List` class then has mutator methods (like adding and removing an element) that **mutate the instance variable**. In our `List` example, these mutator methods create a new sequence representing the modified list, and **store** that new sequence in their mutable instance variable.

Complete the following code (use `cons`):

```
public class List<T> {
  private Sequence<T> contents;
  public List() {
    contents = empty();
  }
  public void prepend(T element) {


  }
}
```

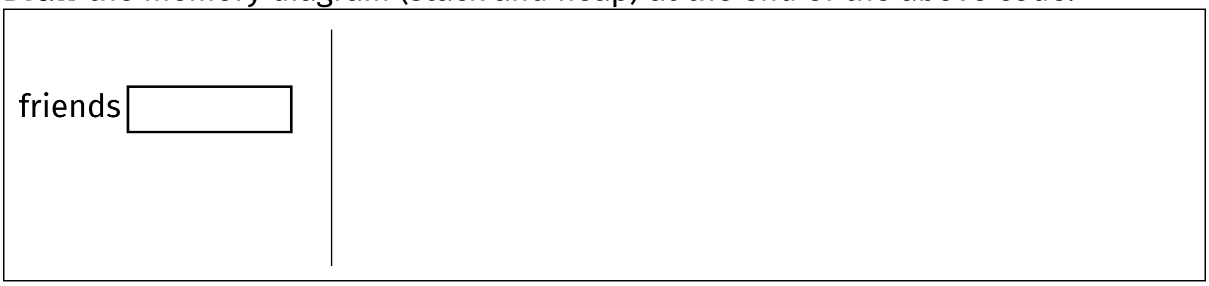
Can you see the mutation? And can you see how the `prepend` method does not need to return the modified list? Because we don't **create a new** list; **we mutate the existing** list!

This design allows us to keep a "stable" reference to a list object. There is one object. It represents a list that can change over time. When we add or remove elements, it is still the **same** list. We don't need to pass around a reference to a new list, because the old and new list are represented by the same `List` object.

```
List<Person> friends = new List<Person>();
// now friends is empty
friends.prepend(new Person(11));
// now friends contains person 11
friends.prepend(new Person(22));
// now friends contains person 11 and person 22
```

The drawback of this design is the general problem of **non-local mutation**: a given `List` object may represent a list with certain contents **at one point in time**, but a list with different content **at another point in time**. Reasoning about our program (for example to find and fix a bug) now requires keeping track of state changes over **time** ("Do you mean my friends ***before* or *after*** I befriended Jim?"). That can be quite challenging. A list at a time is not necessary **equal** to the **same** list later.

Draw the memory diagram (stack and heap) at the end of the above code:



Java Collection Classes: `java.util.List`

The Java **Collection classes** (in package `java.util`) implement all kinds of data structures, such as lists, sets, and maps.

The interface `java.util.List` has various subtypes that provide different kinds of implementations (with different costs and benefits).

Here is a part of the `List` interface and one class that implements it: `LinkedList`. **Draw** a class diagram:

```
public interface List<E> {
  public abstract int size();
  public abstract E get(int index);
  public abstract int indexOf(Object o);
  public abstract E set(int index, E element);
  public abstract void add(E element);
  public abstract E remove(int index);
  //…
}

public class LinkedList<E> implements List<E> {
  public int size() { … }
  //…
}
```

There is one fundamental difference between our `Sequence` class and Java's `List`. It has nothing to do with inheritance. What could that be?

Draw the memory diagrams (stack and heap) at the end of the following scenarios. Assume `public class LinkedList<E> … { private Node<E> first; … }` and `class Node<T> { private T item; private Node<T> next; … }`:

```
List<String> people =
  new LinkedList<String>();
people.add("You");
people.add("Me");
int myIndex =
  people.indexOf("Me");
```

```
Sequence<String> nobody =
  empty();
Sequence<String> justMe = cons("Me", nobody);
Sequence<String> youAndMe = cons("You", justMe);
int myIndex =
  youAndMe.indexOf("Me");
```

An object of type `List` is **mutable**. That is, we can append elements to a list, and the same `List` object that used to be empty is now not empty anymore!

Another Mutable Collection: Map

Sometimes we need to maintain a mapping from a **key** to a **value**. For example, we may need to map names to phone numbers, or countries to population counts. We can represent a mapping with a **data structure** or with a **function**. In both cases, if we get a key, we need to be able to **map** it to the corresponding value.

Mapping from a country to its population count **with a function**:

```
public static int populationByCountry(String country) {
  return
    country.equals("Switzerland") ? 8796669 :
    country.equals("Italy") ? 58870762 :
    country.equals("Germany") ? 83294633 : -1;
}
```

```
int swissPopulation = populationByCountry("Switzerland");
```

Mapping from a country to its population count **with a data structure**:

```
Map<String,Integer> populationByCountry = …;
populationByCountry.put("Switzerland", 8796669);
populationByCountry.put("Italy", 58870762);
populationByCountry.put("Germany", 83294633);
```

```
int swissPopulation = populationByCountry.get("Switzerland");
```

When using a function, we simply call the function. When using a map data structure, we first create the data structure (allocating a Map object, and then `put`ting all the **key-value pairs** in it), and then **look up** a **value** by its **key**.

Mark whether each of the following claims holds or not:

Yes	No	Claim
❏	❏	Encoding a mapping in a conditional (a conditional expression or statement, in a function body) means the mapping is static – it is defined by the developer and **baked into the code** (“hard-coded”).
❏	❏	Encoding a mapping in a data structure means the mapping is dynamic – it could be **changed at runtime** (e.g., `put` on the map).
❏	❏	The **type** of the **key** and the type of the **value** must be the same.

Implementing a Map Class

Implement the `get` method of the following `Map` class (use a `while`-loop):

```
public class Map {
  private Sequence<Pair<K,V>> keyValuePairs;

  public Option<V> get(K key) {









  }
}
```

Java Collection Classes: `java.util.Map`

The interface `java.util.Map` has various subtypes that provide different kinds of implementations (with different costs and benefits).

Here is a simplified version of `Map` and one class that implements it: `TreeMap`. **Draw** a class diagram:

```
public interface Map<K,V> {
  public abstract V get(Object key);
  public abstract V put(K key, V value);
  public abstract V remove(Object key);
  //…
}

public class TreeMap<K,V> implements Map<K,V> {
  public V get(Object key) { … }
  //…
}
```

Using Maps

You want to build a multi-player game. You have a `Player` class to represent a player (including their score, and information about their current status).

At the beginning one can enter the names of all players, and the application will create a `Player` object for each player. Later, during the game, the user can enter the name of a player in the user interface (as a `String`), and the application then needs to find the corresponding `Player` object.

Complete the implementation of the method bodies:

```
public class Game {

  private Map<String,Player> playersByName;

  public Game() {

  }

  public void addPlayer(String name) {
    Player player = new Player();

  }

  public static Player getPlayerByName(String name) {

  }
}
```

Java Collection Classes: `java.util.Set`

Besides lists and maps, the Java library also provides **sets**. The interface `Set` and implementations such as `TreeSet`.

Sets are like lists, but they disallow duplicate elements, and they don't provide indexed access; instead they provide a method `boolean contains(E value)`.