INT103 Midterm Exam 2/2023 9:30-11:30 – 2hrs – April 5, 2024

0. Replace 999 with the last three digits of your student id. You must follow java naming convention, format your code properly, and use only meaningful comments (if any) to get full credits. Submit only 4 files as follows:
   (1) **`Record999Keeper.java`**
   (2) **`NonNull999Keeper.java`**
   (3) **`Real999Keeper.java`**
   (4) **`AssetSystem999.java`**
1. Create a **`Record999Keeper<K,V>`** `interface` in **`keeping999`** `package`. This interface stores key-value pairs (**`K`**, **`V`**), like the **`Map`** `interface` but the classes that implement this interface should not allow **`null`** on keys and values. This interface contains the following **`public`** methods.
   1.1. a **`size`** method that does not receive any argument but returns a value of type **`int`**.
   This method returns the number of key-value pairs that it stores.
   1.2. a **`record`** method that receives two input arguments: **`k`** of type **`K`** and **`v`** of type **`V`** respectively
   and returns a value of type **`boolean`**. This method stores the key-value pair. If success, it returns **`true`**.
   Otherwise, it returns **`false`**.
   1.3. a **`containsKey`** method that receives an input argument: **`k`** of type **`K`** and returns a **`boolean`**.
   This method checks if it contains the key **`k`** or not.
   1.4. a **`contains`** method that receives two input arguments: **`k`** of type **`K`** and **`v`** of type **`V`** respectively,
   and returns a value of type **`boolean`**. This method checks if it contains the key-value pair (**`k`**, **`v`**) or not.
   1.5. an **`update`** method that receives two input arguments: **`k`** of type **`K`** and **`v`** of type **`V`** respectively,
   and returns a value of type **`V`**. This method checks if it contains the key **`k`** or not. If no, it returns **`null`**.
   Otherwise, it replaces the old value of type **`V`** with the new value **`v`** and returns the old value of type **`V`**.
   1.6. a **`retrieve`** method that receives an input argument: **`k`** of type **`K`** and returns a value of type **`V`**.
   This method returns the value of type **`V`** that is paired with the key **`k`**.
   However, if it does not have this key **`k`** in the storage, it returns **`null`** instead.
   1.7. an **`erase`** method that receives an input argument: **`k`** of type **`K`** and returns a value of type **`V`**.
   This method is like the previous method, but it also removes the key-value pair from the storage.

   // the functional programming part //

   1.8. a **`findKey`** method that receives a **`predicate`** of type **`Predicate<K>`** and returns a value of type **`K`**.
   This method uses the **`predicate`** to retrieve and return the value of type **`K`** if it finds the key
   that matches the **`predicate`**.
   1.9. a **`stream`** method that does not receive any argument.
   This method returns a stream of type **`Stream<Map.Entry<K,V>>`**.
2. In the **`keeping999`** `package`, create a `public abstract` **`NonNull999Keeper<K,V>`** `class`
   that **`implements`** the **`Record999Keeper<K,V>`** `interface`. This class contains the following methods.
   2.1. an `abstract protected` **`save`** method that receives two arguments: **`k`** of type **`K`** and **`v`** of type **`V`** respectively
   and returns **`void`**. This method must be overridden by its concrete subclasses to store the key-value pair (**`k`**, **`v`**)
   in its storage.
   2.2. an `@Override public final` **`record`** method that receives two input arguments: **`k`** of type **`K`** and **`v`** of type **`V`**
   respectively and returns a value of type **`boolean`**. This method stores the key-value pair (**`k`**, **`v`**).

If success, it returns **true**. Otherwise, it returns **false**. This method returns **false** if **k** is **null** or **v** is **null**, or it has already **containsKey k** in its storage. Otherwise, it calls the **save** method and returns **true**.

2.3. an `abstract protected` **replace** method that receives two arguments: **k** of type **K** and **v** of type **V** respectively, and returns a value of type **V**. This method must be overridden by its concrete subclasses to replace the old value of type **V** (that is paired with the key **k**) with the new value **v** and returns the old value.

2.4. an `@Override public final` **update** method that receives two input arguments: **k** of type **K** and **v** of type **V** respectively, and returns a value of type **V**. This method returns **null** if **k** is **null** or **v** is **null**, or it does not **containsKey k** in its storage. Otherwise, it calls the **replace** method and returns the return value from the **replace** method.

3. In the **keeping999** package, create a `public` **Real999Keeper<K,V>** `class` that **extends** the **NonNull999Keeper<K,V>** `class`. This class contains the following field and methods.

3.1. a `private final` **notebook999** field of type **Map<K.V>**.

3.2. a **public** empty constructor that initializes **notebook999** to a new **HashMap**.

3.3. an `@Override public` **size** method that returns the size of the **notebook999** which is a **Map**.

3.4. an `@Override public` **containsKey** method that receives a **k** of type **K**.
This method checks if the **notebook999** `Map` **containsKey k** or not and returns the result.

3.5. an `@Override public` **contains** method that receives two input arguments:
k of type **K** and **v** of type **V** respectively and returns a value of type **boolean**.
This method checks if it contains the key-value pair (**k**, **v**) or not.

3.6. an `@Override protected` **save** method that receives two input arguments:
k of type **K** and **v** of type **V** respectively and returns **void**.
This method stores the key-value pair (**k**, **v**) in the **notebook999** `Map`.

3.7. an `@Override protected` **replace** method that receives two input arguments:
k of type **K** and **v** of type **V** respectively and returns a value of type **V**.
For the key **k**, this method replaces the old value of type **V** (that is mapped with the key **k**.
in the **notebook999** `Map`) with the new value **v** (the input argument) and returns the old value of type **V**.

3.8. an `@Override public` **retrieve** method that receives an argument **k** of type **K** and returns a value of type **V**.
This method gets the value of type **V** that is mapped with the key **k** in the **notebook999** `Map`
and returns the result.

3.9. an `@Override public` **erase** method that receives an argument **k** of type **K** and returns a value of type **V**.
This method gets the value of type **V** (that is mapped with the key **k** in the **notebook999** `Map`),
removes this key-value pair (**k**, **v**) from the **notebook999**, and returns the value of type **V**.

// the functional programming part //

3.10. a **public** constructor that receives a **mapSupplier** of type **Supplier<Map<K,V>>**.
This method uses the **mapSupplier** to initialize the value of the **notebook999** `Map`.

3.11. an `@Override public` **findKey** method that receives a **keyPredicate** of type **Predicate<K>**
and returns a value of type **K**. This method uses a **stream** on the key set of the **notebook999** `Map`
to find a key that matches the **keyPredicate**. *This method returns* **null** *if not found*.

3.12. an `@Override public` **stream** method.
This method returns the **stream** on the entry set of the **notebook999** `Map`.

4. Given the **Employee** class and the **Asset** class in the **company** package,
   create a public **AssetSystem999** class in the same package.
   This class stores the information about the asset holder.
   It uses **Record999Keeper<K,V>** to store key-value pairs of assets and employees to note that which asset is held by which employee. This class contains the following field and methods.

   4.1. a private final **keeper999** field of type **Record999Keeper<Asset,Employee>**.

   4.2. an empty constructor that initializes the value of **keeper999** to a new **Real999Keeper**.

   4.3. a public **register** method that receives two arguments: an **a** of type **Asset** and an **e** of type **Employee**, and returns a value of type **boolean**. This method calls the **record** method with **a** and **e** (parameters) respectively to the **keeper999** and returns the result from the **record** method.

   4.4. a public **getHolder** method that receives an argument **a** of type **Asset** and returns a value of type **Employee** who holds the **a** asset. It returns **null** if no one holds that asset.

   4.5. a public **transfer** method that receives three arguments:
   (1) an **a** of type **Asset**
   (2) a **from** of type **Employee**
   (3) a **to** of type **Employee**,
   and returns a value of type **boolean**. This method returns **false** if any argument is **null**, or the **a** asset is not held by the **from** employee. Otherwise, this method calls the **update** method on the **keeper999** with the **a** asset and the **to** employee, and returns **true**.

   4.6. a public **unregister** method that receives two arguments: an **a** of type **Asset** and an **e** of type **Employee** and returns a value of type **boolean**. This method returns **false** if any argument is **null**, or the **a** asset is not held by the **e** employee. Otherwise, this method calls the **erase** method on the **keeper999** with the **a** asset and returns **true**.

   // the functional programming part //

   4.7. a **public** constructor that receives a **keeperSupplier** of type **Supplier<RecordKeeper<Asset,Employee>>**. This method uses the **keeperSupplier** to initializes the value of the **keeper999**.

   4.8. a **public** constructor that receives two arguments:
   (1) a **keeperFunction** of type **Function<Map<Asset,Employee>,RecordKeeper<Asset,Employee>>**
   (2) a **mapSupplier** of type **Supplier<Map<Asset,Employee>>**. This method uses the **keeperFunction** to initialize the value of the **keeper999** and uses the **mapSupplier** to initialize the value of the **Map** in the **keeper999**.

-- -- -- -- -- -- -- the end of the exam questions -- -- -- -- -- -- --

   // the part that should be in the exam //

   4.9. a public **getOwner** method that receives an argument **assetPredicate** of type **Predicate<Asset>** and returns a value of type **Employee** who holds an asset that meets the **assetPredicate**.
   It returns **null** if no one holds that asset. Use the **findKey** method of the **Record999Keeper** for this.

   4.10. a public **getAllAssetOwnedByEmployeeName** method that receives an argument **name** of type **String** and returns a value of type **List<Asset>**. This method finds all assets owned by any employee named **name**.
   It returns a list of all assets owned by the employee(s). If there is no such assets, it returns an empty list.

```
// Employee class ===============================================

package company;

public class Employee {
   private static int nextId=0;
   private final int id;
   private String name;
   private double salary;

   public Employee(String name, double salary) {
      if (name == null || salary <= 0.0)
         throw new IllegalArgumentException();
      id = nextId++;
      this.name = name;
      this.salary = salary;
   }

   public int getId() { return id; }
   public String getName() { return name; }
   public double getSalary() { return salary; }

   @Override
   public int hashCode() { return id; }

   @Override
   public boolean equals(Object obj) { return this == obj; }

   @Override
   public String toString() {
      return "Asset{" + "code=" + id + ", name=" + name + ", salary=" + salary + '}';
   }
}

// Asset class ===============================================
package company;

public class Asset {
   private static int nextCode=0;
   private final int code;
   private final String name;
   private final double price;

   public Asset(String name, double price) {
      if (name == null || price <= 0.0)
         throw new IllegalArgumentException();
      code = nextCode++;
      this.name = name;
      this.price = price;
   }

   public int getCode() { return code; }
   public String getName() { return name; }
   public double getPrice() { return price; }

   @Override
   public int hashCode() { return code; }

   @Override
   public boolean equals(Object obj) { return this == obj; }

   @Override
   public String toString() {
      return "Asset{" + "code=" + code + ", name=" + name + ", price=" + price + '}';
   }
}
```

-- -- -- -- -- -- -- the end of the exam -- -- -- -- -- -- --

-- -- The following pages contain the solution to the exam. -- --

```
// SOLUTION TO INT103 MIDTERM EXAM 2/2023 April 5, 2024, 9:30-11:30 (2 hours)


// (1 of 4) Record999Keeper.java --------------------
package keeping999;

import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public interface Record999Keeper<K,V> {
   public int size();
   public boolean record(K k, V v);
   public boolean containsKey(K k);
   public boolean contains(K k, V v);
   public V update(K k, V v);
   public V retrieve(K k);
   public V erase(K k);

   // functional programming part //
   public K findKey(Predicate<K> predicate);
   public Stream<Map.Entry<K,V>> stream();
}

// the end of Record999Keeper.java --------------------




// (2 of 4) NonNull999Keeper.java --------------------
package keeping999;

public abstract class NonNull999Keeper<K,V> implements Record999Keeper<K,V> {

   abstract protected void save(K k, V v);

   @Override
   public final boolean record(K k, V v) {
      if (k == null || v == null || containsKey(k)) return false;
      save(k, v);
      return true;
   }

   abstract protected V replace(K k, V v);

   @Override
   public final V update(K k, V v) {
      if (k == null || v == null || !containsKey(k)) return null;
      return replace(k, v);
   }
}

// the end of NonNull999Keeper.java --------------------
```

```java
// (3 of 4) Real999Keeper.java --------------------
package keeping999;

import java.util.*;
import java.util.function.*;
import java.util.stream.Stream;

public class Real999Keeper<K,V> extends NonNull999Keeper<K,V> {

   private final Map<K,V> notebook999;

   public Real999Keeper() { notebook999 = new HashMap<>(); }

   @Override
   public int size() { return notebook999.size(); }

   @Override
   public boolean containsKey(K k) { return notebook999.containsKey(k); }

   @Override
   public boolean contains(K k, V v) {
      return v != null && v.equals(notebook999.get(k)); }

   @Override
   public V retrieve(K k) { return notebook999.get(k); }

   @Override
   public V erase(K k) { return notebook999.remove(k); }

   @Override
   protected void save(K k, V v) { notebook999.put(k, v); }

   @Override
   protected V replace(K k, V v) { return notebook999.replace(k, v); }

   // the functional programming part //

   public Real999Keeper(Supplier<Map<K,V>> mapSupplier) {
      notebook999 = mapSupplier.get();
   }

   @Override
   public K findKey(Predicate<K> predicate) {
      return notebook999.keySet()
          .stream()
          .filter(predicate)
          .findAny().orElse(null); // or alternatively, .findFirst().orElse(null);
   }

   @Override
   public Stream<Map.Entry<K,V>> stream() {
      return notebook999.entrySet().stream();
   }
}

// the end of Real999Keeper.java --------------------
// (4 of 4) AssetSystem999.java --------------------
package company;

import java.util.*;
import java.util.function.*;
import keeping999.*;

public class AssetSystem999 {

   private final Record999Keeper<Asset,Employee> keeper999;

   public AssetSystem999() { keeper999 = new Real999Keeper(); }
   public boolean register(Asset a, Employee e) { return keeper999.record(a, e); }
   public Employee getHolder(Asset a) { return keeper999.retrieve(a); }

   public boolean transfer(Asset a, Employee from, Employee to) {
      if (!keeper999.contains(a, from)) return false;
      keeper999.update(a, to);
      return true;
   }

   public boolean unregister(Asset a, Employee e) {
      if (!keeper999.contains(a, e)) return false;
      keeper999.erase(a);
      return true;
   }

   // the functional programming part //

   public AssetSystem(Supplier<RecordKeeper<Asset,Employee>> keeperSupplier) {
      keeper999 = keeperSupplier.get();
   }

   public AssetSystem(
      Function<Map<Asset,Employee>,RecordKeeper<Asset,Employee>> keeperFunction,
      Supplier<Map<Asset,Employee>> mapSupplier) {
      Keeper999 = keeperFunction.apply(mapSupplier.get());
   }

   // the part that should be in the exam //

   public Asset findExpensiveAsset(double expensive) {
      return keeper999.findKey(a -> a.getPrice() > expensive);
   }

   public List<Asset> getAllAssetOwnedByEmployeeName(String name) {
      return keeper999.stream()
         .filter(e -> e.getValue().getName().equals(name))
         .map(e->e.getKey())
         .toList();
   }
}

// the end of AssetSystem999.java --------------------
```

-- -- -- -- -- -- -- the end of the solution to the exam -- -- -- -- -- -- --