Colecciones en Java

Daniel Alejandro Arzate Osorio Manuel Rodriguez Urdapilleta
Juan Pablo Gómez Vázquez

October 2021

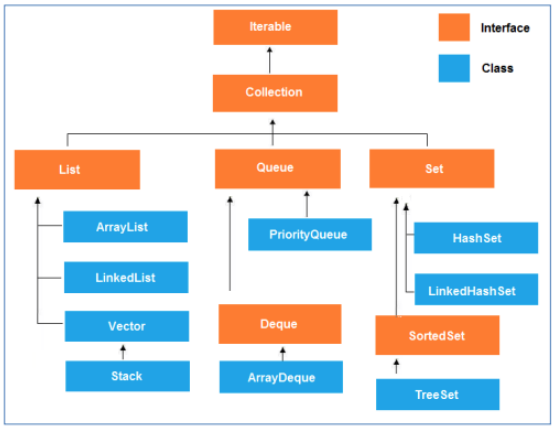
1. Objetivo

Que el alumno conozca los principales aspectos teóricos y prácticos de las colecciones y sus aplicaciones en el lenguaje de programación Java, así mismo que ponga en práctica los conceptos básicos de la programación orientada a objetos.

2. Introducción

Una colección representa un grupo de objetos. Esto objetos son conocidos como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. En Java, se emplea la interfaz genérica **Collection** para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección... ya que se trata de métodos definidos por la interfaz que obligatoriamente han de implementar las subinterfaces o clases que hereden de ella. Partiendo de la interfaz genérica extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

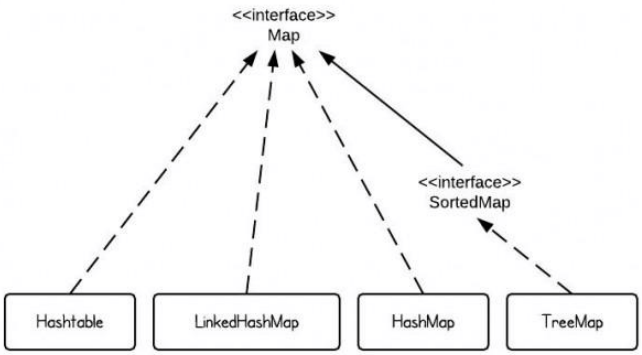
El siguiente esquema resume cómo se organiza la interfaz Collection y sus derivaciones:



Un tipo de colecciones de objetos en Java son los Maps. A pesar de que estas estructuras denominadas "mapas" o "mapeos" son colecciones de datos, en el api de Java no derivan de la interface Collection. No obstante, muchas veces se estudian conjuntamente junto a las clases e interfaces derivadas de Collection.

La interfaz nos dice qué podemos hacer con un objeto. Un objeto que cumple determinada interfaz es "algo con lo que puedo hacer X". La interfaz no es la descripción entera del objeto, solamente un mínimo de funcionalidad con la que debe cumplir.

Como corresponde a un lenguaje tan orientado a objetos, estas clases e interfaces están estructuradas en una jerarquía. A medida que se va descendiendo a niveles más específicos aumentan



los requerimientos y lo que se le pide a ese objeto que sepa hacer.

Interfaz Collection:

Esta interfaz es "la raíz" de todas las interfaces y clases relacionadas con colecciones de elementos. Algunas colecciones pueden admitir duplicados de elementos dentro de ellas, mientras que otras no admiten duplicados. Otras colecciones pueden tener los elementos ordenados, mientras que en otras no existe orden definido entre sus elementos. Java no define ninguna implementación de esta interface y son respectivamente sus subinterfaces las que implementarán sus métodos como son por ejemplo las interfaces Set o List (que son subinterfaces de Collection). Una colección es de manera genérica un grupo de objetos llamados elementos. Esta interfaz por tanto será usada para pasar colecciones de elementos o manipularlos de la manera más general deseada.

Las operaciones básicas de Collection entonces son:

- **add():**
  Añade un elemento.
- **iterator():**
  Obtiene un "iterador" que permite recorrer la colección visitando cada elemento una vez.
- **size():**
  Obtiene la cantidad de elementos que esta colección almacena.
- **contains():**
  Pregunta si el elemento t ya está dentro de la colección.

Hay tres interfaces que extienden de Collection: List, Set y Queue. Cada una de ellas agrega condiciones sobre los datos que almacena, además de ofrecer más funcionalidad.

Interfaz List (Listas):

Una lista es una estructura secuencial, donde cada elemento tiene un índice o posición. El índice de una lista es siempre un entero y el primer elemento ocupa la posición 0. Para trabajar con ellas se utiliza la interfaz **List**. Las implementaciones más recomendables son:

- **ArrayList:** esta es la implementación típica. Se basa en un array redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones.
- **LinkedList:** esta implementación permite que mejore el rendimiento en ciertas ocasiones. Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

A parte de los métodos heredados de Collection, añade métodos que permiten mejorar los siguientes puntos:

- **Acceso posicional a elementos:** manipula elementos en función de su posición en la lista.
- **Búsqueda de elementos:** busca un elemento concreto de la lista y devuelve su posición.
- **Iteración sobre elementos:** mejora el Iterator por defecto.
- **Rango de operación:** permite realizar ciertas operaciones sobre rangos de elementos dentro de la propia lista.

La interfaz List añade varios métodos, entre los que se encuentran principalmente:

- **add(int indice, E e):** Inserta un nuevo elemento en una posición. El elemento que estaba en esta posición y los siguientes pasarán a la siguiente posición.
- **get(int indice):** Devuelve el elemento en la posición especificada.
- **indexOf(E e):** Devuelve la primera posición en la que se encuentra un elemento; -1 si no está.
- **lastIndexOf(E e):** Última posición del el elemento especificado; o -1 si no está.
- **remove(int indice):** Elimina el elemento de la posición indicada.
- **set(int indice, E e):** Pone un nuevo elemento en la posición indicada. Devuelve el elemento que se encontraba en dicha posición anteriormente.

El cuándo usar una implementación u otra de List variará en función de la situación en la que nos encontremos. Generalmente, ArrayList será la implementación que usemos en la mayoría de situaciones. Sobretodo, varían los tiempos de inserción, búsqueda y eliminación de elementos, siendo en unos casos una solución más óptima que la otra.

Interfaz Set (Conjuntos):

La interfaz Set define una colección que no puede contener elementos duplicados. Esta interfaz contiene, únicamente, los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos. Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos **equals** y **hashCode**. Para comprobar si dos Set son iguales, se comprobarán si todos los elementos que los componen son iguales sin importar en el orden que ocupen dichos elementos.

Para definirlos se utiliza la interfaz **Set**, que no añade nuevos métodos de la interfaz Collection. Por lo tanto, con los métodos de la anteriores se pueden manipular los conjuntos.

Dentro de la interfaz Set existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Estas son:

- **HashSet:** Esta implementación almacena los elementos en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Es la implementación más empleada debido a su rendimiento y a que, generalmente, no nos importa el orden que ocupen los elementos. Proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.

- **TreeSet:** Esta implementación almacena los elementos ordenándolos en función de sus valores. Es bastante más lento que HashSet. Los elementos almacenados deben implementar la interfaz Comparable. Esta implementación garantiza, siempre, un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- **LinkedHashSet:** Esta implementación almacena los elementos en función del orden de inserción. Es, simplemente, un poco más lento que HashSet.

El tiempo de inserción es menor en HashSet y mayor en TreeSet. Es importante destacar que la inicialización del tamaño inicial del Set a la hora de su creación es importante ya que, en caso de insertar un gran número de elementos, podrían aumentar el número de colisiones y; con ello, el tiempo de inserción.

Interfaz Queue:

Se conoce como cola a una colección especialmente diseñada para ser usada como almacenamiento temporario de objetos a procesar. Las operaciones que suelen admitir las colas son "encolar", "obtener siguiente", etc. Por lo general las colas siguen un patrón que en computación se conoce como **FIFO** (por la sigla en inglés de "First In - First Out" - "lo que entra primero, sale primero"), lo que no quiere decir otra cosa que lo obvio: El orden en que se van obteniendo los "siguientes" objetos coincide con el orden en que fueron introducidos en la cola.

- **LinkedList:** La diferencia con respecto a la administración de pilas en Java es que para trabajar con colas debemos crear un objeto de la clase LinkedList e implementar la interfaz Queue:

  Queue cola1 = new LinkedList();

  La clase LinkedList implementa la interfaz Queue que es la que declara los método principales para trabajar una cola.
- **PriorityQueue:** Una variante de una cola clásica la implementa la clase PriorityQueue. Cuando se agregan elementos a la cola se organiza según su valor, por ejemplo si es un número se ingresan de menor a mayor.

Los métodos de la interfaz Queue son:

- **add(E e):** Inserta el elemento especificado en esta cola si es posible hacerlo inmediatamente sin violar las restricciones de capacidad, retorna true cuando se realiza correctamente y arrojando un ***IllegalStateException*** si no hay espacio disponible actualmente.
- **element():** Recupera, pero no elimina, el encabezado de esta cola.
- **offer(E e):** Inserta el elemento especificado en esta cola si es posible hacerlo inmediatamente sin violar las restricciones de capacidad.
- **remove():** Recupera y elimina el encabezado de esta cola.

Interfaz Map (Diccionarios o Mapas):

Los diccionarios (también conocidos como mapas o matrices asociativas) son estructuras de datos donde cada elemento tiene asociado una clave que usaremos para recuperarlo (en lugar del índice de una lista). Para definirlos se utiliza la interfaz **Map**. En este caso se trabaja con dos clases una que se utiliza como clave (K) y otra para almacenar los valores (V). La idea es que cada elemento se almacena mediante un par de objetos **(K,V)**. Esta estructura de datos nos permite obtener el objeto V muy rápidamente, a partir de su clave K. Por ejemplo, podríamos almacenar objetos de la clase Vehiculo y utilizar como clave su matrícula en un String. De esta forma, a partir de la matrícula un diccionario encontraría el vehículo asociado muy rápidamente.

Dentro de la interfaz Map existen varios tipos de implementaciones realizadas dentro de Java. Estas son:

- **HashMap:** Esta implementación almacena las claves en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- **TreeMap:** esta implementación almacena las claves ordenándolas en función de sus valores. Es bastante más lento que HashMap.
- **LinkedHashMap:** esta implementación almacena las claves en función del orden de inserción. Es, simplemente, un poco más costosa que HashMap.

Veamos los métodos de la interfaz Map. A diferencia de otras colecciones no hereda del interfaz Collection.

- **put(K key, V value):** Añade un nuevo par clave-valor al diccionario.
- **get(Object key):** Retorna el valor asociado a una clave o null si no se encontró.
- **remove(Object key):** Elimina el par clave-valor que corresponde a la clave.
- **containsKey(Object key):** Comprueba si está la clave especificada.
- **containsValue(Object value):** Comprueba si está el valor.
- **keySet():** Devuelve un conjunto con las claves contenidas en el diccionario.
- **values():** Devuelve una colección con solo los valores del Map.
- **isEmpty():** Comprueba si la colección está vacía.
- **size():** Devuelve el número de elementos que contiene la colección.
- **clear():** Elimina todos los elementos de la colección.

El cuándo usar una implementación u otra de Map variará en función de la situación en la que nos encontremos. Generalmente, HashMap será la implementación que usemos en la mayoría de situaciones. HashMap es la implementación con mejor rendimiento (como se ha podido comprobar en el análisis de Set), pero en algunas ocasiones podemos decidir renunciar a este rendimiento a favor de cierta funcionalidad como la ordenación de sus elementos.

3. Documentación

El proyecto esta dividido en 5 clases: *Hotel, Administrador, Cliente, Habitacion y Reservacion.*

3.1. Hotel

El método principal del programa se encuentra en la clase *Hotel*. Este constiste en un menú con 8 opciones:

1. Registrar Cliente
2. Reservar Habitación
3. Modificar Reservación
4. Mostrar Clientes
5. Mostrar Reservación
6. Mostrar Habitaciones ocupadas

7. Mostrar Habitaciones Disponibles

8. Salir

Todas las opciones se realizan llamando a su respectivo método en la clase *Administrador*.
Al ejecutar el programa, se llama al método *Habitacion.crearHabitaciones()* para simular las habitaciones prestablecidas en el hotel.

3.2. Administrador

Esta clase maneja todas las operaciones generales que se realizan en el menú principal. Por lo tanto, la gran mayoria de los métodos son *public static* pues en el método principal no se crea ninguna instancia de la clase.

3.2.1. *public static void registrarCliente(Scanner sc)*

Este método solo recibe el *Scanner* que se utiliza en todo el programa.
Primero le pregunta al usuario su nombre. Con esa informacíon, llama al constructor *Cliente(String nombre)* para crear un nuevo cliente.

3.2.2. *public static void reservarHabitacion(Scanner sc)*

Este método solo recibe el *Scanner* que se utiliza en todo el programa.
El método primero pregunta por el nombre de algún cliente resgistrado. Si este no existe, le pide al usuario que se registre primero. De lo contrario, primero checa si dicho cliente ya tiene una reservación, ya que un mismo cliente no puede tener mas de una reservación a la vez. De no tener reservación, se llama al método *reservar(Scanner sc, Cliente cliente)* que contiene toda la lógica para asosiar cliente - habitación - reservación.
Una vez realizada la reservación, agrega el objeto de tipo *Reservacion* generado a la lista estática de reservaciones.

3.2.3. *private static void reservar(Scanner sc, Cliente cliente)*

El método comienza creando un nuevo objeto *Reservacion*, llamando a su método constructor vacío.

En seguida se le pide al usuario que ingrese las fechas de llegada y salida de su reservación. Para evitar cualquier problema, se llama al método estático *Reservacion.validarFecha(String fecha, Scanner sc)* y se verifica que la fecha esté en el formato correcto. Además, se verifica que la fecha de salida sea despúes de la de llegada.
El programa le pide al usuario que eliga entre las habitaciones disponibles en las fechas ingresadas. Se verifica el número de habitación y finalmente se realizan las operaciones necesarias para asosiar los tres objetos entre sí:

```
// Reservacion - Cliente
res.setCliente(cliente);

// Cliente - Reservacion
cliente.setReservacion(res);

// Reservacion - Habitacion
res.setHabitacion(habitacion);

// Habitacion - Reservacion
habitacion.getReservaciones().add(res);
```

3.2.4. *public static void modificarReservacion(Scanner sc)*

Este método trabaja de manera similar al método *reservarHabitacion(Scanner sc)*. Sin embargo, este requiere que se ingrese un cliente con una previa reservación, para así, realizar los cambios necesarios llamando al método *reservar(Scanner sc, Cliente cliente)*.

3.2.5. *public static void mostrarClientes()*

Este método itera sobre el Set estático de clientes e imprime en pantalla sus nombres, en caso de que un cliente ya tenga una reservación, se llama al método *imprimirReservación(Cliente cliente)*.

3.2.6. *public static void verReservacion(Scanner sc)*

Este método le pide al usuario que ingrese un cliente registrado y llama al método *imprimirReservación(Cliente cliente)*.

3.2.7. *private static void imprimirReservacion(Cliente cliente)*

El método utiliza al parámetro *cliente* como entrada para acceder a los demás objetos relacionados. Este método llama a los métodos *toString()* de los objetos relacionados con una reservación: *Cliente, Reservacion y Habitacion.*

3.2.8. *public static void habitacionesOcupadas(Scanner sc)*

Le pide al usuario que ingrese una fecha. La valida utilizando *Reservacion.validarFcha(String fecha, Scanner sc)* y llama al método estático *Habitacion.mostrarHabitaciones(String fecha)*.

3.2.9. *public static void habitacionesDisponibles(Scanner sc)*

Le pide al usuario que ingrese una fecha de llegada y una fecha de salida. Las valida utilizando *Reservacion.validarFcha(String fecha, Scanner sc)* y llama al método estático *Habitacion.mostrarHabitaciones(String fechaLlegada, String fechaSalida)*.

3.3. Cliente

En esta clase se implementa la colección de *Set* en forma de un *HashSet*. Se tomó esta decisión pues los clientes son algo que no se podia repetir, solo podia un cliente con el mismo nombre.

Atributos:

- *private String nombre.*
- *private Reservacion reservacion.*
- *static Set ⟨Cliente⟩ clientes.*

Getters:

- *getNombre().*
- *getReservacion().*

Setters:

- *setReservacion(String reservacion).*

3.3.1. *public Cliente(String nombre)*

El constructor de esta clase asigna el atributo *nombre* y agrega la instancia al *HashSet* estático.

3.3.2. *public static Cliente buscarCliente(String nombre)*

Toma como parámetro el nombre de un cliente y realiza una búsqueda lineal en el *HashSet* de clientes. Al encontrar el objeto *Cliente* con nombre *nombre*, lo devuelve, si no lo encuentra, el método devuelve *null*.

3.3.3. *@Override public int hashCode() y public boolean equals(Object obj)*

Estos dos métodos se sobrescribieron para poder ajustar el comportamiento del *HashSet*, ajustando el criterio de igualdad del objeto a solo el nombre. De esta manera, solo puede haber un objeto de tipo *Cliente* con el mismo nombre dentro del Set.

3.4. Habitacion

En esta clase se implementa la colección de *Map* de la forma *Hashtable* y *List* de la forma *ArrayList*.
El *Hashtable* se utilizó para poder tener acceso a los objetos de tipo *Habitacion*, teniendo sólo su número pues cada número del 1 al número de habitaciones disponibles, se mappea a un objeto *Habitacion* diferente.
En cambio, el *ArrayList* se utilizó para guardar objetos de tipo *Reservacion*, ya que una misma habitación, puede tener varias reservaciones, obviamente, en distintas fechas.

Atributos:

- *public static int numHabitaciones.*
- *private int numero.*
- *private ArrayList ⟨Reservacion⟩ reservaciones.*
- *public static Hashtable ⟨Integer, Habitacion⟩ habitaciones.*

Getters:

- *getReservaciones().*
- *getNumero().*

3.4.1. *public Habitacion(int numero)*

El constructor de esta clase asigna el atributo *numero*.

3.4.2. *public static void crearHabitaciones(int numHabitaciones)*

Este método es llamado al inicio del programa y con una estructura *for*, llama al constructor de la clase *numHabitaciones* veces, para el valor del contador como parámetro y agrega la instancia al *Hashtable*.

3.4.3. *public static int validarNumeroHabitacion(Scanner sc, int numHabitaciones)*

Este método le pide al usuario que ingrese el número de una habitación y se asegura de que el parámetro pasado sea el número de una habitación válida. Utiliza una estructura *try-catch* para manejar si se ingresan letras. Al final se devuelve el valor entero validado.

3.4.4. *public static void mostrarHabitaciones()*

Itera sobre los valores del *Hashtable* y llama al método *ptoString()*.

3.4.5. *public static void mostrarHabitaciones(String fecha)*

Llama al método *Reservacion.habitacionesNoDisponibles(String fecha)* para crear un *Hashtable* con las reservaciones que se empalmen en la fecha ingresada. En seguida se imprimen las habitaciones que contengan alguna de estas reservaciones.
Este método se llama en *Administrador.habitacionesOcupadas(Scanner sc)*.

3.4.6. *public static void mostrarHabitaciones(String fechaLlegada, String fechaSalida)*

Llama al método *Reservacion.habitacionesNoDisponibles(String fechaLlegada, String fechaSalida)* para crear un *Hashtable* con las reservaciones que se empalmen en el rango de fechas ingresadas. En seguida se imprimen las habitaciones que NO contengan alguna de estas reservaciones.
Este método se llama en *Administrador.habitacionesDisponibles(Scanner sc)*.

3.5. Reservacion

En esta clase se implementa la clase *LocalDate* y *DateTimeFormatter* para el manejo de fechas.

Atributos:

- *private final static DateTimeFormatter fmt.*
- *static final LocalDate fechaNula.*
- *private LocalDate fechaLlegada.*
- *private LocalDate fechaSalida.*
- *private Habitacion habitacion.*
- *private Cliente cliente.*
- *public static ArrayList ⟨Reservacion⟩ reservaciones.*

Getters:

- *getFechaLlegada().*
- *getFechaSalida().*
- *getHabitacion().*
- *getCliente().*

Setters:

- *setFechaLlegada(String fecha).*
- *setFechaSalida(String fecha).*
- *getHabitacion(Habitacion habitacion).*
- *getCliente(Cliente cliente).*

3.5.1. *public static Hashtable⟨Integer, Reservacion⟩ habitacionesNoDisponibles (String fecha)*

Crea un nuevo *Hashtable*. Itera sobre la lista de reservaciones y, con ayuda de los métodos de la clase *LocalDate*, *isBefore() e isAfter()*, determina si la fecha ingresada se encuentra dentro del rango de alguna reservación, de ser el caso, agrega dicha reservación junto con el número de la habitación que la contiene, al *Hashtable*.

3.5.2. *public static Hashtable⟨Integer, Reservacion⟩ habitacionesNoDisponibles (String fechaLlegada, String fechaSalida)*

Crea un nuevo *Hashtable*. Itera sobre la lista de reservaciones y, con ayuda de los métodos de la clase *LocalDate*, *isBefore() e isAfter()*, determina si el rango de fechas ingresada se empalma con alguna reservación, de ser el caso, agrega dicha reservación junto con el número de la habitación que la contiene, al *Hashtable*.

4. Manual de usuario

4.1. Ejecucion del Programa

1. Para ejecutar el programa, debes desplazarte por la terminal hasta la carpeta donde se encuenren los siguientes archivos:

   - Administrador.java
   - Cliente.java
   - Habitacion.java
   - Reservacion.java

2. Una vez localizado el directorio, procederemos a ejecutarlo con la siguiente sintaxis:

```
javac -encoding utf-8 Hotel.java && java Hotel
```

4.2. Registro

1. En el menu del programa utilizaremos la opcion 1 (Registro de cliente) y daremos enter.
2. Ingresaremos el nombre de la persona para asociarla a la reservación.
3. El cliente ya estara guardado en la 'base de datos'.

NOTA En caso de que se repita un cliente, este no será agreagado de nuevo.

4.3. Hacer una reservación

Es necesario hacer el registro del cliente antes de continuar

1. Una vez realizado el registro del cliente, presionaremos la opcion 2.
2. Ingresaremos el nombre del cliente que realizara la reservación.
3. Ingresaremos la fecha de llegada.
4. Ingresamos la fecha de salida.
5. Se mostrarán las habitaciones disponibles por el momento.
6. Ingresamos la habitacion a reservar.

NOTA Es importante que introduzcan el formato de la fecha adecuado (dd/MM/yy).

4.4. Modificar reservación

1. Para modificar la reservacion, ingresaremos a la opcion 3
2. Ingresaremos el nombre de la persona que hizo la reservación.
3. Nos mostrará la reservacion con el nombre dado.
4. Ingresaremos la fecha de llegada a modificar.
5. Ingresamos la fecha de salida a modificar.
6. Se mostrarán las habitaciones disponibles por el momento.
7. Ingresamos la habitacion a reservar.

NOTA Es importante que introduzcan el formato de la fecha adecuado (dd/MM/yy).

4.5. Visualizacón

4.5.1. Mostrar clientes

1. Al ingresar la opcion 4 podremos visualizar los clientes registrados y si es que han hecho alguna reservación.

4.5.2. Mostrar reservaciones

1. Al ingresar la opcion 5, veremos la reservacion que algún cliente ha realizado en el hotel.
2. Ingresaremos el nombre de la persona que hizo la reservación.
3. La reservación incluira:
   a) Nombre del cliente
   b) Fecha de llegada
   c) Fecha de salida
   d) Habitacion reservada

4.5.3. Mostrar habitaciones ocupadas

1. Al ingresar la opcion 6, nos indicara que ingresemos una fecha para la consulta de habitaciones ocupadas.
2. Se mostrarán todas las habitaciones reservadas en la fecha ingresada, así como los elementos de la reservación.
3. La reservación incluira:
   a) Nombre del cliente
   b) Fecha de llegada
   c) Fecha de salida
   d) Habitacion reservada

4.5.4. Mostrar habitaciones disponibles

1. Al ingresar la opcion 7, nos indicara que ingresemos la fecha de llegada y salida para consultar las habitaciones disponibles.
2. En caso de que alguna habitacion se encuentre ocupada en la el rango de fechas seleccionada, esta no aparecera en la lista.

5. Conclusiones

5.1. Manuel Rodriguez Urdapilleta

Concluyendo, se cumplieron los objetivos del proyecto. Se investigaron y se implementaron las colecciones del lenguaje Java. Además, se utilizaron todos los conceptos de clase vistos hasta el momento para elaborar un trabajo que demostrara y utilizara el uso del paradigma de programación orientado a objetos.

Se cumplió con el trababajo propuesto y se cumplieron todos los puntos en él. Se realizó un análisis y se implementaron las colecciones donde más parecían aportar a un proyecto bien diseñado. Al final se tuvo que investigar aún más sobre problemas que fueron surgiendo en la realización del trabajo (Excepciones, try-catch, @Override, etc.) por lo que es fácil afirmar que este proyecto sí aportó a mis conocimientos.

Este proyecto, además de aportar a nuestro conocimiento de colecciones y el lenguaje Java, me dió una idea de cómo es trabajar en proyectos de programación más grandes, las ventajas que el paradigma orientado a objetos proveen al momento de diseñar software, y distintas estrategias de organización en equipo para trabajos de este tipo.

5.2. Juan Pablo Gómez Vázquez

En este proyecto pude repasar todo lo aprendido hasta el momento, en particular la utilizacion de colecciones no se me dificulto mucho ya que con las practicas y tareas que se han realizado a lo largo del semestre, pude comprender el funcionamiento. El proyecto me gusto porque fue una mezcla de todo, tanto metodos de acceso, colecciones, datos primitivos,etc. Y el equipo trabajo muy bien ya que logramos conectar en las ideas para el desarrollo del codigo.

5.3. Daniel Alejandro Arzate Osorio

El desarrollo del proyecto fue un poco complicado sobre todo en la parte de la documentación, ya que utilizamos un procesador de texto que jamas habíamos utilizado y tuvimos que investigar el funcionamiento del mismo. Por lo demás fue un proyecto bastante enriquecedor, aplicamos los temas vistos en clase tratando siempre de hacerlo correctamente . Se cumplió el objetivo de este proyecto.

6. Referencias

- Oracle.(2020). Recuperado de: https://docs.oracle.com/en/java/javase/16/docs/api/ index.html
- Android Curso. (2017). Recuperado de: http://www.androidcurso.com/index.php/ tutoriales-java-esencial/462-las-colecciones-en-java
- Adictos Al Trabajo. (2015). Recuperado de: https://www.adictosaltrabajo.com/2015/09/25/ introduccion-a-colecciones-en-java/
- Lichtmaierm, N. Recuperado de: http://www.reloco.com.ar/prog/java/collections.html
- Programmer Click. (2020). Recuperado de: https://programmerclick.com/article/1686718375/
- miGabeta. (2017). Recuperado de: https://migabeta.wordpress.com/2016/10/22/api-colecciones-y-genericos-en-java/