Programación Orientada a Objetos III - Uso de colecciones I (listas)

Algoritmos y Estructuras de Datos II

Año 2025

Dr. Pablo Ponzio

Universidad Nacional de Río Cuarto

CONICET

Colecciones

- En casi todo sistema de software es necesario mantener grupos de objetos. Ej.:
    - Todos los estudiantes de la universidad
    - Todos los estudiantes de un curso
    - Las materias que un estudiante está cursando
- Las colecciones son abstracciones que permiten almacenar un número arbitrario de objetos
- Las colecciones usualmente tienen operaciones para:
    - Agregar objetos a la colección
    - Eliminar objetos de la colección
    - Buscar y retornar objetos guardados en la colección
- En esta clase comenzaremos a utilizar colecciones ya implementadas, tomadas de la librería estándar de Java
- Más adelante, implementaremos nuestras propias colecciones y estudiaremos sus características de eficiencia

ArrayList de Java

- `java.util` es el nombre de la biblioteca estándar de Java
  - Implementa varios tipos de colecciones
- `java.util.ArrayList`: implementación de listas de tamaño variable
  - De manera abstracta, `ArrayList` representa secuencias de objetos de tamaño arbitrario: $[o_1, o_2, \ldots, o_N]$
    - Recordar que las secuencias preservan el orden de los elementos
  - Internamente, `ArrayList` implementa las listas con arreglos
    - La implementación se encarga de incrementar el tamaño de los arreglos cuando se excede la capacidad del arreglo actual
    - Esto se hace de manera transparente para el programador
- Como usuarios no necesitamos conocer la implementación, podemos usarla sabiendo que representa secuencias de objetos, y como operan sus métodos sobre estas secuencias

ArrayList: Algunos métodos

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

`ArrayList``()`
Constructs an empty list with an initial capacity of ten.

`boolean`	**`add`**`(E e)` Appends the specified element to the end of this list.
`void`	**`add`**`(int index, E element)` Inserts the specified element at the specified position in this list.
`void`	**`clear`**`()` Removes all of the elements from this list.
`boolean`	**`contains`**`(Object o)` Returns `true` if this list contains the specified element.
`E`	**`get`**`(int index)` Returns the element at the specified position in this list.
`int`	**`indexOf`**`(Object o)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean`	**`isEmpty`**`()` Returns `true` if this list contains no elements.
`E`	**`remove`**`(int index)` Removes the element at the specified position in this list.
`boolean`	**`remove`**`(Object o)` Removes the first occurrence of the specified element from this list, if it is present.
`E`	**`set`**`(int index, E element)` Replaces the element at the specified position in this list with the specified element.
`int`	**`size`**`()` Returns the number of elements in this list.

Ejemplo: MusicOrganizer

- Vamos a implementar un organizador de archivos musicales
- Requisitos de la primera versión:
  - El organizador consiste de una colección de archivos musicales
  - Debe permitir agregar y quitar pistas de la colección
  - Debe poder retornar la cantidad de canciones en la colección
  - Se deben poder listar las canciones almacenadas
  - Para facilitar la implementación de la primera versión, vamos a representar las canciones con sus nombres de archivos (como Strings)
- Más adelante, vamos a ir agregando funcionalidades incrementalmente para mejorar nuestra aplicación
  - Nota: Las metodologías ágiles plantean que el desarrollo de software debe ser incremental: se implementan algunas funcionalidades básicas al inicio, y se van agregando funcionalidades en sucesivas iteraciones hasta llegar a cumplir con las especificaciones

Paquetes

- En Java, podemos organizar grupos de clases en paquetes
- Para acceder a clases de otros paquetes usamos la palabra reservada `import`
  - Ver la primera línea de la figura
- Una vez importada una clase, podemos definir variables del tipo y ejecutar sus métodos
- Más adelante veremos como organizar clases en paquetes

```
import java.util.ArrayList;

/**
 * Stores a sequence of audio files. Files are
 * represented by their filenames (Strings).
 */
public class MusicOrganizer
{
    // An ArrayList for storing the file names of
    private ArrayList<String> files;

    /**
     * @post Create a MusicOrganizer.
     */
    public MusicOrganizer()
    {
        files = new ArrayList<>();
    }
```

Genericidad

- `ArrayList` es una clase genérica: puede almacenar datos de cualquier tipo (objetos)
- Cuando usamos `ArrayList` tenemos que especificar un tipo adicional: el de los objetos a almacenar
  - En la definición del atributo `ArrayList<String>` indica que la lista contendrá `String`s
- En la inicialización del objeto (`new`) tenemos dos opciones:
  - Especificar el tipo explícitamente: `new ArrayList<String>();`
  - Usar el operador `<>`, que infiere el tipo de los elementos de la declaración del atributo: `new ArrayList<>();`
- Los tipos genéricos definen una familia de tipos
  - `ArrayList<String>` y `ArrayList<Person>` son dos tipos diferentes

```java
import java.util.ArrayList;

/**
 * Stores a sequence of audio files. Files are
 * represented by their filenames (Strings).
 */
public class MusicOrganizer
{
    // An ArrayList for storing the file names
    private ArrayList<String> files;

    /**
     * @post Create a MusicOrganizer.
     */
    public MusicOrganizer()
    {
        files = new ArrayList<>();
    }
```

MusicOrganizer: Implementación de métodos

```java
/**
 * @post Add a file to the end of the collection.
 */
public void addFile(String filename)
{
    files.add(filename);
}

/**
 * @post Returns the number of files in the collection.
 */
public int getNumberOfFiles()
{
    return files.size();
}
```

MusicOrganizer: Implementación de métodos

```java
/**
 * @post Add a file to the end of the collection
 */
public void addFile(String filename)
{
    files.add(filename);
}
```

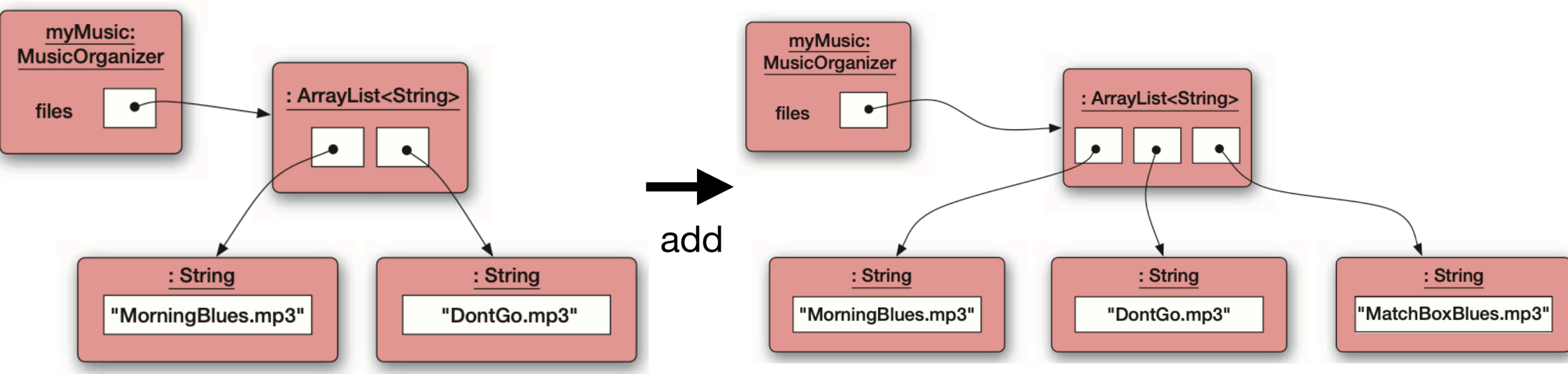
```java
/**
 * @post Returns the number of files in the coll
 */
public int getNumberOfFiles()
{
    return files.size();
}
```

```java
/**
 * @pre 0 <= 'index' < getNumberOfFiles()
 * @post Prints the name of the file in position 'index'
 * to the terminal.
 */
public void listFile(int index)
{
    if(index >= 0 && index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
    }
}
```

```java
/**
 * @pre 0 <= 'index' < getNumberOfFiles()
 * @post Removes the file in position 'index' from the
 * collection.
 */
public void removeFile(int index)
{
    if(index >= 0 && index < files.size()) {
        files.remove(index);
    }
}
```

ArrayList: Diagrama de Objetos

- Para entender como funciona una `ArrayList` es útil graficar los diagramas de objetos a medida que ejecutamos métodos de `MusicOrganizer`



- `ArrayList` funciona de manera abstracta como una secuencia: mantiene el orden en que se insertan los elementos, y permite accederlos en ese orden
- `ArrayList` va incrementado su capacidad a medida que se requiere, de manera transparente al usuario

MusicOrganizer: Algunas conclusiones

- La clase `MusicOrganizer` es muy simple, todo el trabajo difícil lo hace `ArrayList`
- Esta es la gran ventaja de usar bibliotecas: alguien invirtió tiempo y esfuerzo en implementar un módulo, y tenemos acceso a las funcionalidades del mismo
- Que un módulo sea fácil de usar se debe a que usamos abstracción: nos abstraemos de los detalles de implementación de `ArrayList`, y la usamos en base a la especificación de sus métodos
- Una observación importante es que `MusicOrganizer` usa la funcionalidad de `ArrayList` para llevar la cuenta de la cantidad de canciones en la colección
  - Podríamos haber definido nuestra propia cuenta en `MusicOrganizer`, pero esto habría sido duplicar información ya existente
- Duplicar funcionalidades (y código) es una mala práctica de programación y debe evitarse tanto como sea posible
  - Las distintas implementaciones pueden ser inconsistentes y dar lugar a errores
  - Tenemos más código que mantener y entender

MusicOrganizer v2

- Requisitos de la segunda versión: Debe tener todas las funcionalidades de la anterior, y además debe reproducir los archivos musicales
- Usaremos una biblioteca para reproducir .mp3s
- Como es costumbre, nos vamos a abstraer de los detalles de implementación de la biblioteca y la vamos a usar en base a su especificación
- Vamos a generar la documentación de nuestras clases a automáticamente a partir de los comentarios en el código fuente
- Esto es posible porque usamos un formato predefinido para escribirla: Una variante del lenguaje de especificación Javadoc para Java [0]
- Para generar la documentación con gradle ejecutamos: `./gradlew javadoc`
  - Podemos explorar la documentación abriendo el archivo: `docs/index.html`

Constructors

Constructor	Description
`MusicPlayer()`	Constructor for objects of class MusicFilePlayer

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`void`	`playSample(String filename)`	Play a part of the given file.
`void`	`startPlaying(String filename)`	Start playing the given audio file.
`void`	`stop()`	

[0] https://docs.oracle.com/en/java/javase/24/docs/specs/man/javadoc.html

Ejemplo: MusicOrganizer v2

- Agregamos un atributo de tipo `MusicPlayer`, y lo inicializamos en el constructor
- Mejoramos también la precondición de `addFile`, para asegurar que cualquier archivo agregado exista en disco y que es un .mp3 válido

```java
/**
 * Stores a sequence of audio files. Files are
 * represented by their filenames (Strings). It can
 * reproduce the audio files.
 */
public class MusicOrganizer
{
    // An ArrayList for storing the file names of music
    private ArrayList<String> files;
    // A player for the music files.
    private MusicPlayer player;

    /**
     * @post Create a MusicOrganizer.
     */
    public MusicOrganizer()
    {
        files = new ArrayList<>();
        player = new MusicPlayer();
    }

    /**
     * @pre 'filename' exists and is a valid .mp3 file.
     * @post Add a file to the end of the collection.
     */
    public void addFile(String filename)
    {
        files.add(filename);
    }
```

Ejemplo: MusicOrganizer v2

- Implementaremos dos nuevos métodos en la clase, para darle la funcionalidad de reproducir archivos de audio

```
/**
 * @pre 0 <= 'index' < getNumberOfFiles() &&
 *      the track in position 'index' is a valid .mp3.
 * @post Start playing the file in position 'index'.
 */
public void startPlaying(int index)
{

}

/**
 * @post Stop the player.
 */
public void stopPlaying()
{

}
```

Ejemplo: MusicOrganizer v2

- Implementaremos dos nuevos métodos en la clase, para darle la funcionalidad de reproducir archivos de audio

```
/**
 * @pre 0 <= 'index' < getNumberOfFiles() &&
 *      the track in position 'index' is a valid .mp3.
 * @post Start playing the file in position 'index'.
 */
public void startPlaying(int index)
{
    String filename = files.get(index);
    player.startPlaying(filename);
}

/**
 * @post Stop the player.
 */
public void stopPlaying()
{
    player.stop();
}
```

Ejemplo: Main para MusicOrganizer v2

```
public static void main(String[] args) throws InterruptedException {
    MusicOrganizer organizer = new MusicOrganizer();
    String filepath = "audio/BlindBlake-EarlyMorningBlues.mp3";
    organizer.addFile(filepath);
    organizer.startPlaying(0);
    TimeUnit.SECONDS.sleep(5);
    organizer.stopPlaying();
}
```

MusicOrganizer v3

- Requisitos de la tercera versión: Tener todas las funcionalidades de la versión anterior más:
  - Imprimir por consola la lista completa de canciones
  - Imprimir por consola la lista de canciones que matchean con un String dado
  - Retornar el índice de la primera canción que matchea con un String dado
- Vamos a implementar los métodos de la figura

```
/**
 * @post Print the list of all the files in the collection.
 */
public void listAllFiles()
```

```
/**
 * @post Print the names of files matching 'searchString'.
 */
public void listMatching(String searchString)
```

```
/**
 * @post Find the index of the first file matching
 *   'searchString'. Return the index of the first
 *   occurrence, or -1 if no match is found.
 */
public int findFirst(String searchString)
```

Iteración en Java

- Hay tres sentencias iterativas en Java:

```
for (inicialización; condición; acción post cuerpo) {
    cuerpo
}
```

```
while (condición) {
    cuerpo
}
```

```
do {
    cuerpo
} while (condición)
```

- Vamos a resolver nuestro problema usando algunas de ellas

MusicOrganizer v3: Implementación

```
/**
 * @post Print the list of all the files in the collection.
 */
public void listAllFiles()
{
    for (int i = 0; i < files.size(); i++) {
        String filename = files.get(i);
        System.out.println(filename);
    }
}
```

```
/**
 * @post Print the names of files matching 'searchString'.
 */
public void listMatching(String searchString)
{
    for (int i = 0; i < files.size(); i++) {
        String filename = files.get(i);
        if(filename.contains(searchString)) {
            // A match.
            System.out.println(filename);
        }
    }
}
```

MusicOrganizer v3: Implementación

- En este punto la implementación empieza a mostrar sus limitaciones
- Un buen organizador de música debería permitir búsquedas más avanzadas, como buscar por artista, o por nombre de canción
- Sin embargo, debido a que representamos las canciones con un único `String`, este tipo de búsquedas son difíciles de soportar en la implementación actual
- Para obtener un código más limpio que soporte estas nuevas operaciones, vamos a agregar una clase `Track` que modele las pistas y su información: artista, titulo, nombre de archivo, etc.

```
/**
 * @post Find the index of the first file matching
 *   'searchString'. Return the index of the first
 *   occurrence, or -1 if no match is found.
 */
public int findFirst(String searchString)
{
    int index = 0;
    // Record that we will be searching until a match is found.
    boolean searching = true;

    while(searching && index < files.size()) {
        String filename = files.get(index);
        if(filename.contains(searchString)) {
            // A match. We can stop searching.
            searching = false;
        }
        else {
            // Move on.
            index++;
        }
    }
    if(searching) {
        // We didn't find it.
        return -1;
    }
    else {
        // Return where it was found.
        return index;
    }
}
```

MusicOrganizer v4

- Nuevos requisitos de la cuarta versión:
  - Listar las canciones cuyo título matchee un `String` dado
  - Buscar el índice de la primera canción cuyo título matchee el `String` dado
  - Listar las canciones cuyo artista matchee un `String` dado
- Primero vamos a refactorizar el código existente
  - Refactorizar consiste en modificar el código para mejorar algún aspecto del mismo (diseño, eficiencia, comprensibilidad, etc.) preservando el comportamiento de la versión anterior
  - Los tests deben seguir pasando luego de una refactorización

MusicOrganizer v4

- Los pasos a seguir son:
  - Agregar una clase `Track`, y hacer que nuestra colección lleve una lista de `Track`s (en lugar de `String`s)
  - Modificar todos los métodos que lo requieran para que funcionen con la nueva colección
  - Cambiar `listMatching` y `findFirst` para que busquen por autor
  - Implementar un nuevo método `listByArtist`

MusicOrganizer v4: Track

- `this` sirve para hacer referencia al objeto sobre el que se ejecuta un método
  - Por ejemplo, en el cuerpo del constructor:
    - `artist` hace referencia al parámetro formal (enmascara al atributo con el mismo nombre)
    - `this.artist` se usa para hacer referencia al atributo artist
- Cuando no es necesario (no hay un parámetro formal que enmascare a un atributo) usualmente se omite `this`

```java
/**
 * Store the details of a music track,
 * such as the artist, title, and file name.
 */
public class Track
{
    // The artist.
    private String artist;
    // The track's title.
    private String title;
    // Where the track is stored.
    private String filename;

    /**
     * Create a track with the given data
     */
    public Track(String artist, String title, String filename)
    {
        this.artist = artist;
        this.title = title;
        this.filename = filename;
    }
```

MusicOrganizer v4: Track

```
/**
 * Store the details of a music track,
 * such as the artist, title, and file name.
 */
public class Track
{
    // The artist.
    private String artist;
    // The track's title.
    private String title;
    // Where the track is stored.
    private String filename;

    /**
     * Create a track with the given data
     */
    public Track(String artist, String title, String filename)
    {
        this.artist = artist;
        this.title = title;
        this.filename = filename;
    }
```

```
    /**
     * @post Return the artist.
     */
    public String getArtist()
    {
        return artist;
    }

    /**
     * @post Return the title.
     */
    public String getTitle()
    {
        return title;
    }

    /**
     * @post Return the file name.
     */
    public String getFilename()
    {
        return filename;
    }

    /**
     * @post Return details of the track as a String.
     */
    public String getDetails()
    {
        return artist + ": " + title + "  (file: " + filename + ")";
    }
}
```

MusicOrganizer v4: Refactor

```java
/**
 * Stores a sequence of audio tracks. It allows
 * to reproduce the tracks in the collection, and
 * to search for tracks under different criteria.
 */
public class MusicOrganizer
{
    // An ArrayList for storing music files.
    private ArrayList<Track> tracks;
    // A player for the music files.
    private MusicPlayer player;

    /**
     * Create an empty MusicOrganizer
     */
    public MusicOrganizer()
    {
        tracks = new ArrayList<>();
        player = new MusicPlayer();
    }
```

```java
/**
 * @pre 'track' represents a valid .mp3 file.
 * @post Add a track to the end of the collection.
 */
public void addTrack(Track track)
{
    tracks.add(track);
}
```

```java
/**
 * @post Returns the number of tracks in the collection.
 */
public int getNumberOfTracks()
{
    return tracks.size();
}
```

```java
/**
 * @pre 0 <= 'index' < getNumberOfTracks()
 * @post Prints the details of the track in position 'index'
 * to the terminal.
 */
public void listTrack(int index)
{
    if(index >= 0 && index < tracks.size()) {
        Track track = tracks.get(index);
        System.out.println(track.getDetails());
    }
}
```

MusicOrganizer v4: Refactor

```java
/**
 * Stores a sequence of audio tracks. It allows
 * to reproduce the tracks in the collection, and
 * to search for tracks under different criteria.
 */
public class MusicOrganizer
{
    // An ArrayList for storing music files.
    private ArrayList<Track> tracks;
    // A player for the music files.
    private MusicPlayer player;

    /**
     * Create an empty MusicOrganizer
     */
    pub
    {


    }
```

```java
/**
 * @pre 'track' represents a valid .mp3 file.
 * @post Add a track to the end of the collection.
 */
public void addTrack(Track track)
{
    tracks.add(track);
}

/**
 * @post Returns the number of tracks in the collection.
 */
public int getNumberOfTracks()
{
    return tracks.size();
}
```

No sólo modificamos el código sino también lo fuimos mejorando: renombramos variables para darles nombres más apropiados, mejoramos la descripción de las especificaciones, etc.

```
n 'index'
public void listTrack(int index)
{
    if(index >= 0 && index < tracks.size()) {
        Track track = tracks.get(index);
        System.out.println(track.getDetails());
    }
}
```

MusicOrganizer v4: Refactor

```java
/**
 * Stores a sequence of audio tracks. It allows
 * to reproduce the tracks in the collection, and
 * to search for tracks under different criteria.
 */
public class MusicOrganizer
{
    // An ArrayList for storing music files.
    private ArrayList<Track> tracks;
    // A player for the music files.
    private MusicPlayer player;

    /**
     * Create an empty MusicOrganizer
     */
    pub
    {


    }
```

```java
/**
 * @pre 'track' represents a valid .mp3 file.
 * @post Add a track to the end of the collection.
 */
public void addTrack(Track track)
{
    tracks.add(track);
}

/**
 * @post Returns the number of tracks in the collection.
 */
public int getNumberOfTracks()
{
    return tracks.size();
}
```

No sólo modificamos el código sino también lo fuimos mejorando: renombramos variables para darles nombres más apropiados, mejoramos la descripción de las especificaciones, etc.

El objetivo es que en cada iteración el código quede mejor que antes

MusicOrganizer v4: Refactor

```
/**
 * @post Print the list of tracks in the collection.
 */
public void listAllTracks()
{
    for (int i = 0; i < tracks.size(); i++) {
        Track track = tracks.get(i);
        System.out.println(track.getDetails());
    }
}
```

```
/**
 * @post Print the names of the tracks with title
 *   matching 'searchString'.
 */
public void listByTitle(String searchString)
{
    for (int i = 0; i < tracks.size(); i++) {
        Track track = tracks.get(i);
        String title = track.getTitle();
        if(title.contains(searchString)) {
            // A match.
            System.out.println(track.getDetails());
        }
    }
}
```

```
/**
 * @post Print the names of the tracks with title
 *   matching 'artist'.
 */
public void listByArtist(String artist)
{
    for (int i = 0; i < tracks.size(); i++) {
        Track track = tracks.get(i);
        String art = track.getArtist();
        if(art.contains(artist)) {
            // A match.
            System.out.println(track.getDetails());
        }
    }
}
```

```
/**
 * @pre 0 <= 'index' < getNumberOfTracks()
 * @post Removes the track in position 'index' from the
 * collection.
 */
public void removeTrack(int index)
{
    if(index >= 0 && index < tracks.size()) {
        tracks.remove(index);
    }
}
```

MusicOrganizer v4: Refactor

```
/**
 * @post Find the index of the first track with title
 *   matching 'searchString'. Return the index of the
 *   first occurrence, or -1 if no match is found.
 */
public int findFirstByTitle(String searchString)
{
    int index = 0;
    // Record that we will be searching until a match is found.
    boolean searching = true;

    while(searching && index < tracks.size()) {
        Track track = tracks.get(index);
        String title = track.getTitle();
        if(title.contains(searchString)) {
            // A match. We can stop searching.
            searching = false;
        }
        else {
            // Move on.
            index++;
        }
    }
    if(searching) {
        // We didn't find it.
        return -1;
    }
    else {
        // Return where it was found.
        return index;
    }
}
```

```
/**
 * @pre 0 <= 'index' < getNumberOfTracks() &&
 *      the track in position 'index' is a valid .mp3.
 * @post Start playing the track in position 'index'.
 */
public void startPlaying(int index)
{
    Track track = tracks.get(index);
    String filename = track.getFilename();
    player.startPlaying(filename);
}

/**
 * @post Stop the player.
 */
public void stopPlaying()
{
    player.stop();
}
```

Iteradores

- Un iterador es un objeto que permite obtener uno a uno los elementos de una colección
  - En el caso de `ArrayList`, que representa secuencias $[o_1, o_2, ..., o_N]$ de elementos, el iterador va a devolver los elementos en orden: $o_1, o_2, ..., o_N$
  - La clase `java.util.Iterator` define iteradores en Java, y las colecciones predefinidas en `java.util` implementan estos iteradores
  - `java.util.Iterator` define los siguientes métodos [1]:

`boolean`	**`hasNext`**`()` Returns `true` if the iteration has more elements.
`E`	**`next`**`()` Returns the next element in the iteration.
`default void`	**`remove`**`()` Removes from the underlying collection the last element returned by this iterator (optional operation).

[1] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Iteradores

- La forma de iterar sobre los elementos de una colección usando `Iterator` se muestra a continuación:

```
Iterator<ElementType> it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next element
    do something with that element
}
```

- Iteramos mientras haya un elemento siguiente (`it.hasNext() == true`)
- Si existe el elemento siguiente, `it.next()` nos permite obtenerlo
- En un ejemplo concreto:
  - Como tracks es de tipo `ArrayList<Track>`, el iterador irá retornando objetos de tipo `Track`
  - Es por esto que el tipo del iterador es `Iterator<Track>`

```
public void listAllTracks()
{
    Iterator<Track> it = tracks.iterator();
    while(it.hasNext()) {
        Track t = it.next();
        System.out.println(t.getDetails());
    }
}
```

Iteradores

- Los iteradores también nos permiten eliminar elementos de la colección
- En particular, `it.remove()` elimina de la colección el último elemento retornado por `it.next()`
- Por ejemplo, el siguiente código elimina las pistas de un artista dado (`artistToRemove`):

```
Iterator<Track> it = tracks.iterator();
while(it.hasNext()) {
    Track t = it.next();
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        it.remove();
    }
}
```

Iteradores

- Una de las ventajas de los iteradores es que simplifican la iteración sobre los elementos de las colecciones, con la siguiente variante del ciclo `for`:

```
for (ElementType element : collection) {
    loop body
}
```

  - Esto sólo funciona si `collection` implementa `Iterator`

- Por ejemplo, como `ArrayList` implementa `Iterator`, podemos ciclar sobre la colección como a continuación:

```
public void listAllFiles()
{
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

MusicOrganizer v5

- En la próxima versión de nuestro `MusicOrganizer` debemos agregar la siguiente funcionalidad:
  - `MusicOrganizer` debe poder cargar todas las canciones de una carpeta dada
  - Podemos asumir que las canciones están almacenadas en archivos cuyo nombre respeta el siguiente formato: `artista-titulo.mp3`
  - Vamos a usar la clase `TrackReader`, provista como biblioteca, que provee la siguiente funcionalidad:

Constructors

Constructor	Description
`TrackReader()`	Create the track reader, ready to read tracks from the music library folder.

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`ArrayList<Track>`	`readTracks(String folder, String suffix)`	Read music files from the given library folder with the given suffix.

MusicOrganizer v5: Implementación

```java
/**
 * Stores a sequence of audio tracks. It allows
 * to reproduce the tracks in the collection, and
 * to search for tracks under different criteria.
 */
public class MusicOrganizer
{
    // An ArrayList for storing music files.
    private ArrayList<Track> tracks;
    // A player for the music files.
    private MusicPlayer player;
    // A reader that can read music files and loa
    private TrackReader reader;

    /**
     * Create an empty MusicOrganizer
     */
    public MusicOrganizer()
    {
        tracks = new ArrayList<>();
        player = new MusicPlayer();
        reader = new TrackReader();
        readLibrary("../audio");
        System.out.println("Music library loaded. " + getNumberOfTracks() + " tracks.");
        System.out.println();
    }
```

```java
    /**
     * @pre The format of all the filenames in 'folderName'
     *   is: artist-title.mp3
     * @post Read the tracks in 'folderName' and loads all of
     *   them in the organizer.
     */
    private void readLibrary(String folderName)
    {
        ArrayList<Track> tempTracks = reader.readTracks(folderName, ".mp3");
        // Put all the tracks into the organizer.
        for(Track track: tempTracks) {
            addTrack(track);
        }
    }
```

MusicOrganizer v5: Main

```
public static void main(String[] args) {
    MusicOrganizer organizer = new MusicOrganizer();
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter the index of the song to start playing, "+
            "or 's' to stop playing, or 'q' to quit.");

    while (true) {
        String input = scanner.next();
        if (input.equals("q")) {
            organizer.stopPlaying();
            break;
        } else if (input.equals("s")) {
            organizer.stopPlaying();
        } else {
            int index = Integer.parseInt(input);
            organizer.startPlaying(index);
        }
    }

}
```

Actividades

- Leer el capítulo 4 del libro "*Objects First with Java A Practical Introduction using BlueJ*". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016
- Leer el capítulo 6 del libro "Program Development in Java - Abstraction, Specification, and Object-Oriented Design". B. Liskov & J. Guttag. Addison-Wesley. 2001

Bibliografía

- "*Objects First with Java A Practical Introduction using BlueJ*". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016
- "*Program Development in Java - Abstraction, Specification, and Object-Oriented Design*". B. Liskov & J. Guttag. Addison-Wesley. 2001