Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

PRÁCTICA 10: Colecciones

1. OBJETIVO

El objetivo de esta práctica es presentar los conceptos relacionados con colecciones en java. Descargue de la plataforma virtual el código ejemplo, necesario para realizar la práctica.

2. CÓDIGO PROPORCIONADO

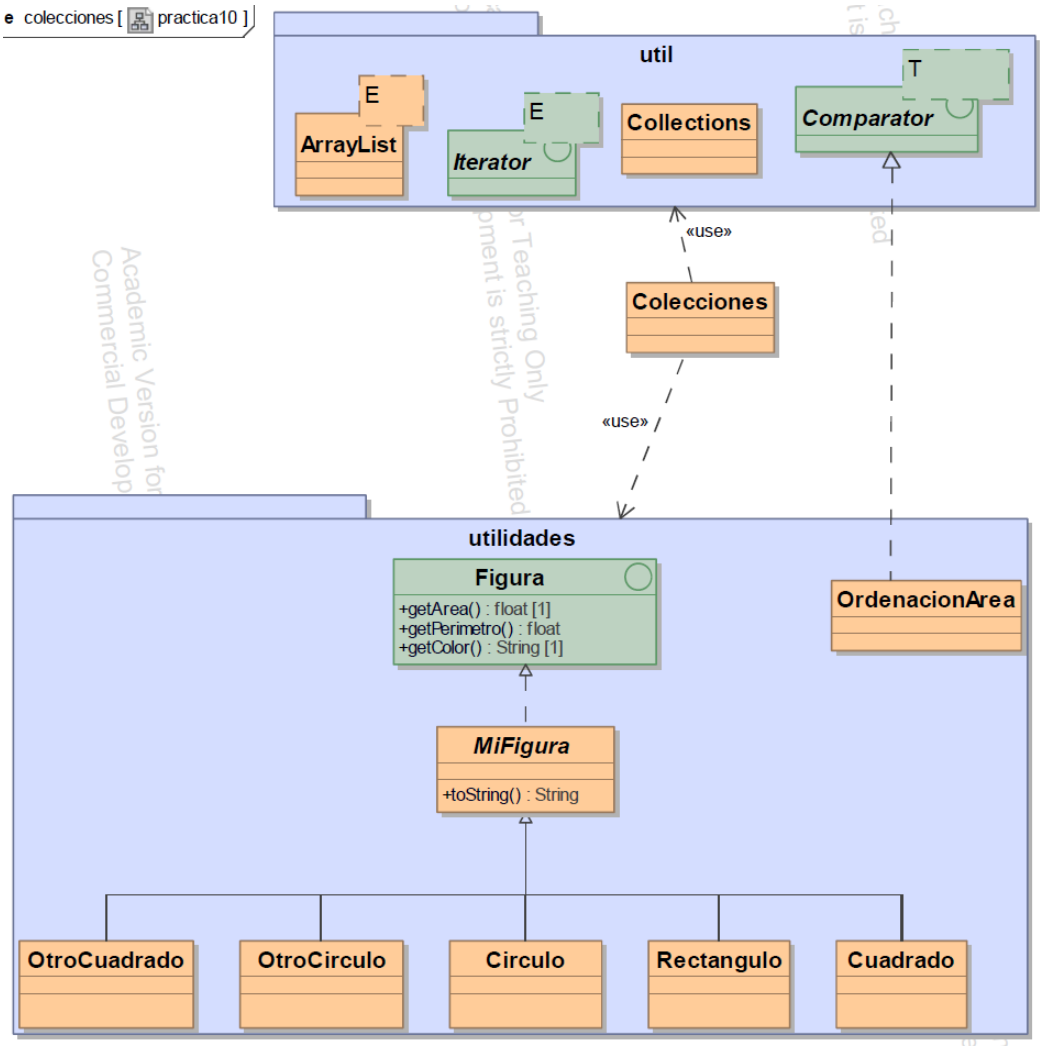

Figura 1. Diagrama de clase para la realización de la práctica.

El diagrama de clases del código proporcionado se puede observar en la Figura 1, que incluye además las clases utilizadas de Java correspondientes a las colecciones. En el fichero descargado, una vez descomprimido, se puede encontrar:

1) Fichero makefile para compilar y ejecutar el código.
2) Fichero makeJavadoc para generar la documentación en html.

```
make –f makeJavadoc
```

3) Carpeta "utilidades", que contiene una serie de clases auxiliares para la realización de la práctica y que se detallan a continuación:

- Interfaz `Figura`: indica la firma de métodos para usar figuras geométricas genéricas, puede verla en Código 1.

Fichero: Figura.java

```
package fp2.poo.practica10.utilidades;
public interface Figura {
    float getArea();
    float getPerimetro();
    String getColor();
}
```

Código 1. Definición de la interfaz `Figura`.

- Clase abstracta `MiFigura`. Observe en Error: No se encuentra la fuente de referencia como implementa `Figura` pero deja sus métodos declarados como abstractos, de modo que son sus subclases las que deben implementar los métodos de `Figura`. Se consigue que todas las implementaciones de `Figura` que hereden de `MiFigura` tengan sobreescrito el método `toString` (de la clase `Object`) para convertir los objetos figura a un `String`. Esto facilita la depuración y análisis del código ejemplo, ya que al imprimir por pantalla el objeto se muestra el `String` que devuelve este método. `getClass` devuelve un objeto de tipo `Class` (que se corresponde con la clase del objeto objetivo), el método `getName` de la clase `Class` devuelve como un `String` el nombre de la clase, incluyendo la ruta completa de paquetes. Observe el mecanismo que se utiliza para imprimir por pantalla únicamente el nombre de la clase (sin incluir los paquetes), se recurre a los métodos de `String`; `lastIndexOf` y `substring`, analice su comportamiento, consulte la clase `String` en la documentación de Oracle si lo necesita [1].

Fichero: MiFigura.java

```
package fp2.poo.practica10.utilidades;
public abstract class MiFigura implements Figura {
  public String toString(){
    int indicepunto=this.getClass().getName().lastIndexOf(".");
    return this.getClass().getName().substring(indicepunto+1)
           +" de color "+this.getColor()+"\n";
  }
  abstract public float  getArea();
  abstract public float  getPerimetro();
  abstract public String getColor();
}
```

Código 2. Definición de la clase abstracta *`Mifigura`* sobreescribe el método *`toString`* de la clase *`Object`* (no implementa los métodos de la interfaz `Figura`, los deja como abstractos, serán sus subclases las que los implementen).

- Subclases de la clase abstracta *`Mifigura`*: varias subclases de la clase anterior (*`Mifigura`*) para representar distintos tipos de figuras geométricas

(`Circulo, OtroCirculo, Cuadrado, OtroCuadrado, Rectangulo`). Todas las subclase implementan los métodos de la interfaz `Figura`.

- `OrdenacionArea`: implementación de la interfaz `Comparator[2]` que permitirá comparar y ordenar figuras geométricas en función de su área. Observe la construcción de este comparador "personalizado" que compara figuras por su área en Código 3. Posteriormente verá cómo este comparador permitirá ordenarlas de mayor a menor (lo verá en Código 14). Este diseño de la librería permite la creación de algoritmos personalizados a los requisitos del programador facilitando además su reutilización. Observe en Código 3 el uso del método `compareTo` del envolvente `Float` (todos los envolventes lo incluyen ya que implementan la interfaz `Comparable`).

Fichero: OrdenacionArea.java

```
package fp2.poo.practica10.utilidades;
…
public class OrdenacionArea implements Comparator<Figura>{
     public int compare(Figura f1, Figura f2) {
         float a1=f1.getArea();
         float a2=f2.getArea();
         Float area1=new Float(a1);
         Float area2=new Float(a2);
         return area2.compareTo(area1);
    }
}
```

Código 3. Implementación de un comparador de figuras personalizado. Clase `OrdenacionArea`

4) Carpeta "colecciones", que contiene la clase principal, en la que se usa la clase `ArrayList[3]`, como ejemplo de colección.

3. INTRODUCCIÓN A LAS COLECCIONES

Una **colección** es un objeto que agrupa múltiples **elementos** en una sola unidad, por ello a veces se conoce también como contenedor, puede verse como una agregación de objetos (ya que cada **elemento** es a su vez un objeto). **El entendimiento y manejo de colecciones está muy relacionado con los conceptos de estructuras dinámicas estudiados en la asignatura "Fundamentos de programación I"**.

El paquete `java.util` incluye el entorno de trabajo (Framework) denominado "Collections" [4,5] (Java Collections Framework), que proporciona clases que permiten organizar y manipular colecciones de objetos, de cualquier tipo, de una forma unificada y estandarizada. Esta API proporciona interfaces para gestionar estructuras dinámicas, independientemente de la clase a la que pertenezcan los objetos de la estructura, así como implementaciones de las mismas (listas para usar). De este modo, si se reutilizan las clases ya proporcionadas, el esfuerzo de programación para manejar colecciones de objetos se reduce, ya que estas clases nos simplifican mucho la manipulación de las mismas (abstraen de la gestión de memoria, proporcionan métodos para la inserción y recuperación de los **elementos**, proporcionan algoritmos útiles, como los de ordenación…), optimizando además la eficiencia de los programas. Y por otro lado, si se necesita crear nuevas funcionalidades para la gestión de colecciones, más personalizadas a nuestras necesidades (nuevos algoritmos de

ordenación, métodos de búsqueda…) se facilita que otros puedan reutilizar nuestro código siempre que se implemente conforme a las interfaces propuestas.

De modo que resumiendo, el hecho de tener este entorno estandarizado facilita la interoperatividad y reutilización, reduciendo además el esfuerzo de diseño, de desarrollo y de aprendizaje de nuevas interfaces de programación de aplicaciones (API).

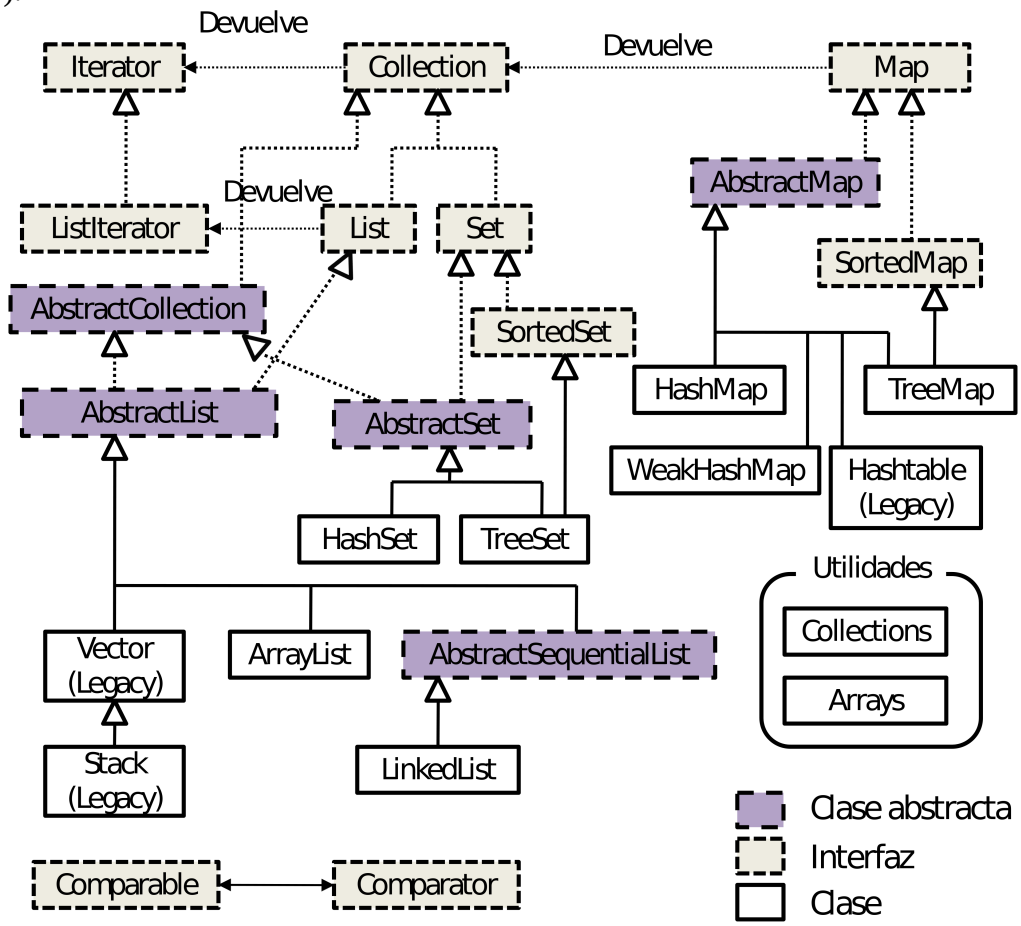

Figura 2. Elementos principales de la API para la gestión de colecciones de objetos.

El entorno de trabajo, que podemos entender como una API y cuyos elementos principales puede verse en la Figura 2, incluye:

- 14 interfaces de colecciones, base para representar diferentes tipos de colecciones (conjuntos, listas, mapas…).
- Implementaciones (realizaciones) básicas de las interfaces.
- Implementaciones abstractas, parciales, que facilitan a los programadores la realización de sus propias implementaciones.
- Implementaciones heredadas de versiones previas de Java (como `Vector` y `Hashtable`).
- Implementaciones de propósito específico, para situaciones particulares o con características optimizadas (uso concurrente, sincronización, alto rendimiento…).
- Implementaciones diseñadas especialmente para facilitar el uso concurrente.
- Implementaciones de tipo “envoltorio”, que añaden funcionalidades a las más básicas.

- Algoritmos, en forma de métodos estáticos que proporcionan funciones útiles y reutilizables para la gestión de colecciones (como por ejemplo algoritmos de ordenación de listas). Son polimórficos, el mismo método puede utilizarse en diferentes implementaciones de interfaz.
- Interfaces de apoyo para las de colecciones.
- Funciones útiles para el manejo de arreglos de primitivas y referencias de objetos.

La interfaz principal es `Collection<E>` [6], de ella heredan `Set`, `List`, `SortedSet`, `NavigableSet`, `Queue`, `Deque`, `BlockingQueue` y `BlockingDeque`.

Las otras interfaces `Map`, `SortedMap`, `NavigableMap`, `ConcurrentMap` and `ConcurrentNavigableMap` no heredan de `Collection`, ya que representan "mapas" (estructuras de pares clave/valor con las que trabajará la próxima práctica) en lugar de auténticas colecciones, aunque sí contienen operaciones que permiten manipularlas como si fueran colecciones.

Además de estas interfaces se incluyen implementaciones, como se ha comentado anteriormente. La tabla que se muestra a continuación incluye sólo las clases más utilizadas y generales.

		Implementaciones				
		Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**	**Hash Table + Linked List**
Interfaces	**Set**	`HashSet`		`TreeSet`		`LinkedHashSet`
	List		`ArrayList`		`LinkedList`	
	Deque		`ArrayDeque`		`LinkedList`	
	Map	`HashMap`		`TreeMap`		`LinkedHashMap`

Tabla 1. Principales interfaces y realizaciones del entorno

4. LA INTERFAZ COLLECTION<E> Y SUS DERIVADAS

`Collection<E>` [6] es una interfaz genérica, de modo que el tipo concreto de objeto contenido se especifica cuando se declara la instancia de una implementación de esta interfaz, o más bien de alguna de sus subinterfaces más específicas (como `Set` o `List`), ya que no se proporcionan implementaciones directas de `Collection`. Puede ver un ejemplo de declaración en el código proporcionado (clase principal, `Colecciones`). En este caso se utiliza la implementación `ArrayList` de la interfaz `List`, especificando que los elementos de la colección serán implementaciones de la interfaz `Figura`. Esto se muestra en el siguiente ejemplo de código:

```
List<Figura> serieDeFiguras = new ArrayList<Figura>();
```

Cada subinterfaz de `Collection` especifica unas características determinadas en la gestión de sus elementos (ordenación, elementos duplicados…), y por tanto **es importante elegir la implementación que más se adapte a las necesidades del desarrollo.**

La interfaz `Collection` declara métodos básicos de gestión de una colección, por ejemplo:

- `int size()` o `boolean isEmpty()`: para conocer aspectos de su tamaño. Puede ver ejemplos de su uso en el código proporcionado en:

```
if(serieDeFiguras.isEmpty())
   System.out.println("NO hay elementos en la coleccion");
else
   System.out.println("SI hay elementos en la coleccion, concretamente "
                      +serieDeFiguras.size()+"\n");
```

- `boolean add(E element)` o `boolean remove(Object element)`: para añadir o eliminar un elemento de la colección. Puede ver ejemplos de su uso en el código proporcionado, en:

```
Figura cuad3 = new OtroCuadrado (8.9f,"SERENITY");
serieDeFiguras.add (cuad3);
serieDeFiguras.remove(cuad3);
```

- `boolean contains(Object element)`: para verificar si `element` es un elemento de la colección. Puede ver ejemplo de su uso en el código proporcionado en:

```
if(serieDeFiguras.contains(cuad3))
      System.out.println(cuad3
             + " SI pertenece a la coleccion y esta en la posicion "
            + serieDeFiguras.indexOf(cuad3)
            + "\n\n");
else
      System.out.println(cuad3+" NO pertenece a la coleccion\n\n");
```

- `Iterator<E> iterator()`: para proporcionar un iterador que permite moverse por la colección. Puede ver un ejemplo de obtención y uso de un iterador en el ejemplo proporcionado en:

```
Iterator<Figura> iterador=serieDeFiguras.iterator();
while(iterador.hasNext()){
      System.out.println(iterador.next());
}
```

La clase `Collection` incluye además operaciones que operan en la colección completa. Lanzan una excepción del tipo `NullPointerException` si la colección o el objeto que se les proporciona son nulos (`null`). Algunos ejemplos son:

- `boolean containsAll(Collection<?> c)`: devuelve `true` si la colección objetivo contiene todos los elementos en la colección especificada como argumento.
- `boolean addAll(Collection<? extends E> c)`: añade todos los elementos de la colección especificada como argumento en la objetivo.
- `boolean removeAll(Collection<?> c)`: elimina de la colección objetivo todos los elementos que estén también contenidos en la especificada como argumento.
- `boolean retainAll(Collection<?> c)`: elimina de la colección objetivo todos los elementos que no estén contenidos en la especificada como argumento. Así que mantiene sólo aquellos que estén contenidos en c.
- `void clear()`: elimina todos los elementos de la colección.

`addAll`, `removeAll`, y `retainAll` devuelven `true` si la colección objetivo se modificó al ejecutar el método.

También se incluyen métodos que proporcionan conversiones de colecciones a matrices, trasladando los elementos de una colección a una matriz. Principalmente se utilizan para hacer de puente con APIs que necesiten matrices como entrada.

- `Object[] toArray()`: los elementos de la colección objetivo se meten en un nuevo array de objetos con un tamaño idéntico al número de elementos de la misma. Ejemplo de invocación: `Object[] a = c.toArray();` (los elementos de la colección `c` , objetivo, se meten en `a`)
- `<T> T[] toArray(T[] a)`:los contenidos de la colección objetivo se trasladan a una nueva matriz con elementos de la clase T, cuya longitud es idéntica al número de elementos de la colección objetivo. Ejemplo de invocación: `String[] a = c.toArray(new String[0]);` (los elementos de la colección `c`, objetivo, se meten en `a`)

Por último, se proporcionan algoritmos genéricos de ordenación, búsqueda, etc, como se puede observar en el ejemplo en el que se utiliza un algoritmo, proporcionado por el método estático `reverse` de la clase `Collections[7]`, que permite cambiar el orden de los elementos de la colección, dándoles la vuelta:

```
System.out.println("Orden original \n"+serieDeFiguras);
Collections.reverse(serieDeFiguras);
System.out.println("\nNuevo Orden \n"+serieDeFiguras);
```

Y facilidades para que se puedan crear otros y reutilizarlos de forma sencilla y extensa. Como se puede observar en el ejemplo proporcionado, en el que se ha realizado una implementación de la clase `Comparator` (a la que hemos llamado `OrdenacionArea`) que permite comparar figuras por el tamaño de su área y que se reutiliza para ordenar las figuras de la colección en función de ésta utilizando el método estático `sort` de la clase `Collections`. Observe que se utiliza `compareTo`, método disponible en los envolventes de tipos genéricos, que permite comparar dos objetos de ese tipo. Esto es porque estos envolventes (wrappers) implementan la interfaz `Comparable`. Puede ver el código de ejemplo a continuación:

Fichero: OrdenacionArea.java

```
public class OrdenacionArea implements Comparator<Figura>{
    public int compare(Figura f1, Figura f2) {
        float a1=f1.getArea();
        float a2=f2.getArea();
        Float area1=new Float(a1);
        Float area2=new Float(a2);
        return area2.compareTo(area1);
    };
}
```

```
OrdenacionArea ordenador=new OrdenacionArea();
Collections.sort(serieDeFiguras,ordenador);

System.out.println(
                "\nAhora ordenadas por Area, de mayor a menor \n");

for (Figura tmp: serieDeFiguras) {
     System.out.println(tmp+": Area= "+tmp.getArea());
}
```

Las subinterfaces derivadas de `Collection` son:

- `Set`: una colección que no puede contener elementos duplicados. Se usa para representar conjuntos (las cartas de una baraja, las asignaturas en las que está matriculado un alumno…)

- `SortedSet`: es un tipo particular de conjunto en el que los elementos se gestionan en orden ascendente. Se incluyen algunas operaciones adicionales que sacan partido de esa ordenación. Se utilizan normalmente cuando los elementos del conjunto tienen esta naturaleza ordenada, por ejemplo un listado de palabras (orden alfabético) o los socios de un club (según su número de socio).

- `List`: una colección ordenada, o secuencia. Puede incluir elementos duplicados. Se utiliza normalmente cuando se necesita un control preciso de la posición de los elementos, y se usa la misma como un índice (entero) para acceder a los elementos. Esta interfaz es muy similar a la interfaz `Vector`, de las primeras versiones de java. En esta práctica trabajamos con una implementación concreta de `List` (`ArrayList`).

- `Queue`: una colección que se utiliza principalmente para almacenar elementos antes de su procesado. Además de las operaciones básicas sobre colecciones una cola proporciona operaciones adicionales de inserción, extracción e inspección. Habitualmente los elementos se ordenan tal y como llegan, de modo que se extraen en el mismo orden de entrada (FIFO, first-in, first-out), pero esta no es la única posibilidad, por ejemplo se pueden tener colas con prioridad, que ordenan elementos según un comparador proporcionado. Cualquiera que sea el criterio de ordenación se entiende que la cabecera de la cola es el elemento que se recuperará cuando se realice una solicitud de recuperación. Así en una cola FIFO cada nuevo elemento se inserta en la última posición, mientras que en otras se pueden insertar en diferentes posiciones. Cada implementación concreta de una cola debe especificar sus propiedades de ordenación.

NOTA: En telecomunicaciones estos conceptos son especialmente relevantes debido a que los algoritmos de gestión de colas de mensajes en los equipos de comunicación influyen notablemente en las características de los sistemas de comunicación.

- `Deque`: de forma similar a la anterior se utiliza principalmente para almacenar elementos antes de su procesado y añade operaciones a las básicas. Se pueden utilizar tanto con ordenación FIFO como en modo pila (LIFO, last-in, first-out), de modo que cada nuevo elemento se puede insertar, recuperar o eliminar de cualquiera de los dos extremos.
- `Map`: es un objeto que permitirá relacionar claves unívocas con valores, por lo que las claves no pueden estar duplicadas y sólo pueden identificar un valor. Es muy similar a `Hashtable` de las primeras versiones de Java.
- `SortedMap`: es un tipo particular de mapa que mantiene sus elementos en orden ascendente según su clave. Usados cuando los pares clave/valor de la colección tienen esa naturaleza ordenada, como un diccionario o un listín telefónico.

En las últimas versiones de Java existen tres métodos para recorrer colecciones:

1) Usando operaciones de agregación (que queda fuera de los contenidos tratados en esta práctica)
2) Con bucles for-each
3) Usando Iteradores (`Iterators`)

En la versión 7, la utilizada en las prácticas, sólo se pueden utilizar las dos últimas. Previamente vio un ejemplo de uso de un iterador, el uso del bucle for-each lo puede ver en el ejemplo en:

```
System.out.println("Listado de la coleccion:\n\n");

for (Figura fig : serieDeFiguras) {
      System.out.println(fig);
}
```

5. EJERCICIOS

Debe haber descomprimido el fichero proporcionado y analizado detenidamente las clases proporcionadas, puede comprobar que la Figura 1 representa la estructura del código proporcionado.

Mantenga abierto el fichero "Colecciones" que contiene la clase principal para analizar qué está ocurriendo en el ejemplo.

A continuación ejecute `make`, que compilará y ejecutará el código ejemplo proporcionado. Si quiere disponer de la documentación en html también puede ejecutar `make –f makeJavadoc.`

Cuando la ejecución pare, esperando un return/intro, analice qué ha ocurrido y qué código se corresponde a ese comportamiento. Vuelva a repetir la operación cada vez que realice un ejercicio propuesto y haya modificado el código.

- Observe la declaración de la colección, `serieDeFiguras` será una referencia a una colección de tipo `ArrayList`, de elementos que serán implementaciones/realizaciones de la interfaz `Figura`. Observe también el uso del método `isEmpty` para verificar que al crearse está vacía.

```
/*
 * Este ejemplo utiliza una colección de tipo ArrayList,
 * que es una implementación de la interfaz List.
 */
 List<Figura> serieDeFiguras = new ArrayList<Figura>();

/*******************************************************
 ********COMPROBACIÓN DE QUE ESTÁ VACÍA: isEmpty   *******
 *******************************************************/
 System.out.println("\n************************************");
 System.out.println( "Se ha creado la coleccion (ArrayList)");
 System.out.println( "Se comprueba que esta vacia.       ");
 System.out.println("************************************\n");
 if(serieDeFiguras.isEmpty())
      System.out.println("NO hay elementos en la coleccion.\n");
 else
      System.out.println("SI hay elementos en la coleccion.\n");
```

Código 4. Creación de un `ArrayList` de elementos de tipo `Figura` y uso del método `isEmpty` en el `main` de la aplicación (clase Colecciones).

- A continuación observe como crear figuras, añadirlas a la colección y verificar el tamaño. ¿Qué métodos se han utilizado? Piense ¿Qué

diferencia práctica cree que habrá entre realizar el `new` previamente y luego usar la referencia al objeto en el `add`, como se hace con `cuad1, 2 y 3`, o incluir el `new` en el método de añadir, como se hace con el resto de figuras? ¿Cuándo usaría una u otra? Observe como puede imprimirse `serieDeFiguras` con el método `println`.

```
/*******************************************************
 ********AÑADIR ELEMENTOS A LA COLECCIÓN: add    *********
 *******************************************************/
   /*
 * Se crean dos objetos del tipo Cuadrado y se añaden a la lista.
 */
Figura cuad1 = new Cuadrado (3.5f,Color.ORANGE);
Figura cuad2 = new Cuadrado (2.2f,Color.YELLOW);
serieDeFiguras.add (cuad1);
serieDeFiguras.add (cuad2);

/*********************************************************
 ******** COMPROBACIÓN DE QUE NO ESTÁ VACÍA: isEmpty *******
 ******** SE MUESTRA EL NÚMERO DE ELEMENTOS: size   ********
 *********************************************************/

System.out.println("\n***********************************");
System.out.println( "EJEMPLO: acabo de meter dos elementos,");
System.out.println( "Se comprueba que no esta vacia.       ");
System.out.println("***********************************\n");

if(serieDeFiguras.isEmpty())
       System.out.println("NO hay elementos en la coleccion");
else
       System.out.println("SI hay elementos en la coleccion, concretamente "
                       + serieDeFiguras.size()+"\n");
try{
       System.out.println("Pulse intro para continuar");
       System.in.read();
}catch(IOException exc){
    System.err.println(exc+"Error al leer\n\n");
}

/*
 * Se crea un nuevo cuadro del tipo OtroCuadrado y se añade a la lista.
 */
Figura cuad3 =       new OtroCuadrado (8.9f,"AMARILLO SERENIDAD");
serieDeFiguras.add (cuad3);

/*
 * Se crea un círculo del tipo Circulo y se añade a la lista.
 */
serieDeFiguras.add (new Circulo (3f, Color.BLACK));


/*
 * Se crea un círculo del tipo OtroCirculo y se añade a la lista.
 */
serieDeFiguras.add (new OtroCirculo (4f,"MELOCOTON MADURO"));

/*
 * Se crean dos rectángulos del tipo Rectangulo y se añaden a la lista.
 */
serieDeFiguras.add (new Rectangulo (2f, 3f, "CUARZO ROSA"));
serieDeFiguras.add (new Rectangulo (12f, 3f,"LILA GRIS"));
```

Código 5. Instanciación de distintos tipos de figuras e inserción en la lista.

EJERCICIO 1: INSTANCIE 3 FIGURAS MÁS; UNA DEL TIPO RECTÁNGULO DE COLOR NEGRO, UNA DEL TIPO OTROCÍRCULO DE COLOR NEGRO Y OTRA DEL TIPO OTROCUADRADO DE COLOR BLANCO. AÑÁDALAS A LA COLECCIÓN Y MUÉSTRELA POR PANTALLA

- A continuación observe como puede comprobar si un objeto pertenece a la colección y la primera posición en la que aparece. ¿En qué posición aparece cuad3?

```
System.out.println("\n*****************************************************");
System.out.println( "EJEMPLO: Se busca un objeto usando su referencia,  *");
System.out.println( "         y se muestra su posicion.                  *");
System.out.println("*****************************************************\n");

if(serieDeFiguras.contains(cuad3))
    System.out.println(cuad3
                       + " SI pertenece a la coleccion y esta en la posicion "
                       + serieDeFiguras.indexOf(cuad3)+"\n\n");
else
    System.out.println(cuad3 + " NO pertenece a la coleccion\n\n");
```

Código 6. Usar la referencia de un elemento para verificar si está y saber la posición de la primera aparición. Método principal de la clase `Colecciones.`

- Ahora observe como borrar un elemento de la colección.

```
System.out.println("\n*****************************************************");
System.out.println(  "EJEMPLO: Se borra el elemento usando su referencia, *");
System.out.println(  "         y se vuelve a buscar.                      *");
System.out.println(  "*****************************************************\n");

serieDeFiguras.remove(cuad3);
if(serieDeFiguras.contains(cuad3))
     System.out.println(cuad3
                 + " SI pertenece a la coleccion y esta en la posicion "
                 + serieDeFiguras.indexOf(cuad3)+"\n\n");
else
     System.out.println(cuad3+" NO pertenece a la coleccion\n\n");
```

Código 7. Borrar elementos usando su referencia. Método principal de la aplicación, clase `Colecciones.`

EJERCICIO 2: VUELVA A AÑADIR CUAD3 Y REPITA LA OPERACIÓN DE BÚSQUEDA Y LOS MISMOS MENSAJES ANTERIORES. OBSERVE LA POSICIÓN DE LA FIGURA ANTES Y DESPUÉS ¿QUÉ HA OCURRIDO?¿Quién ocupa ahora la posición 2? ¿Qué conclusiones obtiene?

- Observe que un `ArrayList` puede contener objetos duplicados y que existe un método para mostrar la última ocurrencia de un objeto.

```
System.out.println("\n*******************************************************");
System.out.println(  "EJEMPLO: Elementos duplicados.                         *");
System.out.println(  "         -Se vuelve a insertar cuad3,                  *");
System.out.println(  "          el objeto esta dos veces                     *");
System.out.println(  "*******************************************************\n");

serieDeFiguras.add (cuad3);
if(serieDeFiguras.contains(cuad3)) {
    System.out.println(cuad3
                       + " esta en la posicion "+serieDeFiguras.indexOf(cuad3)
                       + " y tambien esta en la posicion "
                       + serieDeFiguras.lastIndexOf(cuad3) + "\n\n");
}else{
    System.out.println(cuad3+" NO pertenece a la coleccion\n\n");
}
```

Código 8. Mostrar la última ocurrencia de un elemento usando su referencia. Método principal de la aplicación.

- Observe como recorrer la colección con el bucle for-each, imprimiendo por pantalla las figuras. ¿Con qué formato se muestra cada figura? Observe cómo se sobrescribió el método `toString` en la clase `Mifigura`.

```
System.out.println("\n*******************************************************");
System.out.println(  "EJEMPLO: Uso de bucle for-each, para imprimir       *");
System.out.println(  "         las figuras por pantalla.                    *");
System.out.println(  "*******************************************************\n");

System.out.println("Listado de la coleccion:\n");
for (Figura fig : serieDeFiguras) {
      System.out.print("\t"+ fig);
}
System.out.println();
```

Código 9. Recorrer una lista con el bucle for-each. Método principal de la aplicación.

```
public String toString(){
    int indicepunto=this.getClass().getName().lastIndexOf(".");
    return this.getClass().getName().substring(indicepunto+1)
          + " de color "
          + this.getColor()
          + "\n";
}
```

Código 10. Sobreescritura del método `toString` en la clase `Mifigura.`

EJERCICIO 3: MUESTRE POR PANTALLA UNICAMENTE LOS COLORES DE LA COLECCIÓN, UTILIZANDO EL BUCLE FOR-EACH.

- Observe cómo puede hacer el cálculo del área total de la colección, usando el bucle `for-each`. Observe la característica de polimorfismo. Se invoca siempre igual al método `getArea` pero cada vez se ejecuta un código diferente, según el tipo concreto de la referencia `tmp`.

```
System.out.println("\n*******************************************************");
System.out.println(  "EJEMPLO 2: Uso de bucle for-each, para calcular     *");
System.out.println(  "           el area total.                            *");
System.out.println(  "*******************************************************\n");

float areaTotal = 0.f;

for (Figura tmp: serieDeFiguras) {
    areaTotal = areaTotal + tmp.getArea();
}

System.out.println ( "Tenemos un total de " + serieDeFiguras.size()
                   + " figuras y su area total es de "
                   + areaTotal + " uds cuadradas");
```

Código 11. Uso del bucle for each para calcular el área total. Método principal de la aplicación.

EJERCICIO 4: CACULE EL PERIMETRO TOTAL DE LA COLECCION UTILIZANDO UN BUCLE FOR EACH Y MUÉSTRELO POR PANTALLA

- A continuación observe cómo recorrer la colección con un iterador, mostrando cada figura por pantalla.

```
System.out.println("\n*******************************************************");
System.out.println(  "EJEMPLO: Se obtiene un iterador y se muestran       *");
System.out.println(  "         los objetos.                                *");
System.out.println(  "*******************************************************\n");

Iterator<Figura> iterador=serieDeFiguras.iterator();
while(iterador.hasNext()){
       System.out.print(iterador.next());
}
```

Código 12. Uso del iterador en el método principal de la aplicación.

EJERCICIO 5: MUESTRE POR PANTALLA EL ÁREA DE CADA FIGURA NEGRA DE LA COLECCIÓN USANDO UN ITERADOR Y CUÉNTELAS, MOSTRANDO POR PANTALLA LA CANTIDAD TOTAL DE FIGURAS NEGRAS.

- Ahora observe el uso de algoritmos genéricos (funciones estáticas de la clase `Collections`). En este caso el método usado invierte el orden de una colección. Fíjese bien ¿cómo se realiza la invocación al método reverse? Recuerde las características de los métodos estáticos o de clase.

```
System.out.println("\n*******************************************************");
System.out.println(  "EJEMPLO: uso el algoritmo sort para dar la vuelta   *");
System.out.println(  "         a la colección.                             *");
System.out.println(  "*******************************************************\n");

System.out.println("Orden original \n"+serieDeFiguras);
Collections.reverse(serieDeFiguras);
System.out.println("\nNuevo Orden \n"+serieDeFiguras);
```

Código 13. Uso de la función estática reverse de la clase `Collections` en el método principal de la aplicación.

- Observe la construcción (Código 3) y uso (Código 14) de un comparador "personalizado" para comparar figuras por su área y ordenarlas de mayor a menor área. Este diseño de la librería permite la creación de algoritmos personalizados a los requisitos del programador facilitando además su reutilización.

```
System.out.println("\n*****************************************************");
System.out.println(  "EJEMPLO: Uso de comparador personalizado.           *");
System.out.println(  "         Realiza la ordenacion por area             *");
System.out.println(  "         de mayor a menor.                          *");
System.out.println(  "*****************************************************\n");

OrdenacionArea ordenador=new OrdenacionArea();
Collections.sort(serieDeFiguras,ordenador);
System.out.println("\nAhora ordenadas por Area, de mayor a menor \n");
for (Figura tmp: serieDeFiguras) {
    System.out.println(tmp+": Area= "+tmp.getArea());
}
```

Código 14. Uso del algoritmo de comparación implementado en `OrdenacionArea` en la función estática `sort` de la clase `Collections` (método principal de la aplicación).

EJERCICIO 6: CREE UN COMPARADOR PARA ORDENAR POR **PERÍMETRO DE MENOR A MAYOR.** UTILÍCELO PARA ORDENAR LA COLECCIÓN Y MUÉSTRELA POR PANTALLA, MOSTRANDO EL PERÍMETRO DE CADA FIGURA.

Trabajo a Entregar

Comprima la carpeta que contiene todo el código correspondiente a la práctica, que incluya los ejercicios realizados con el makefile correspondiente para realizar la compilación y ejecución, en un fichero .zip de nombre su uvus. Entregue el fichero en la tarea correspondiente.

6. REFERENCIAS

[1] https://docs.oracle.com/javase/7/docs/api/java/lang/String.html
[2] https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html
[3] https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html
[4] http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html
[5] http://docs.oracle.com/javase/tutorial/collections/index.html
[6] http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html
[7] https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html
[8] https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html