Capítulo 7

Ultimas novedades Java 8, 9, 10 y 11

7.1. Introducción

https://es.scribd.com/doc/19475538/Java-Su-Historia-Ediciones-Versiones-y-Cara

Java aparece en 1995 como un nuevo lenguaje de programación con soporte multiplataforma desarrollado por James Gosling y Sun MicroSystems.

Versión 1.0: La primera versión del lenguaje contiene las clases principales , la maquina virtual y el API gráfico de AWT.

Versión 1.1: Aparece en 1997 e incorpora al lenguaje varias clases que faltaban como Readers /Writers ,Calendars y Bundles . Pero sin ningún lugar a duda su mayor aportación es la inclusión del estandar de JavaBeans y el API de **JDBC** para conexión a bases de datos. Este último supone un salto importante en cuando a la utilización del lenguaje.

Versión 1.2: En 1998 aparece otra evolución importante con la llegada del framework de **Collections** y el API de **Swing** que permite desarrollar interfaces de ventanas más complejos.

Versión 1.3: Avances pequeños en cuanto a APIs , se añade soporte JNDI (La Interfaz de Nombrado y Directorio Java (Java Naming and Directory Interface) es una Interfaz de Programación de Aplicaciones (API) de Java para servicios de directorio. Permite a los clientes descubrir y buscar objetos y datos a través de un nombre. Como todas las APIs de Java que hacen de interfaz con sistemas host, es independiente de la implementación subyacente. Adicionalmente, especifica una

interfaz de proveedor de servicio (SPI) que permite que las implementaciones del servicio de directorio sean integradas en el framework. Las implementaciones pueden hacer uso de un servidor, un fichero, o una base de datos; la elección depende del desarrollador.). Sin embargo el avance en cuento a la arquitectura de la maquina virtual es importante ya que aparece la máquina HotSpot con compilación JIT (Just-in Time).

Versión 1.4: Se produce un salto importante en cuanto a nuevas APIs. Se incorpora un fuerte soporte de XML , **Expresiones Regulares** , Criptografía etc.

Versión 1.5: También denomina Java 5 se producen dos saltos importantes a nivel del core del lenguaje. Por una parte la inclusión de tipos Genéricos que se echaban en falta en el mundo de las colecciones. Por el otro lado la inclusión del concepto de metadatos con el uso de anotaciones. Se amplía el soporte de APIs orientadas a programación concurrente.

Versión 1.6: Esta versión contiene avances muy puntuales con la inclusión de un API de compilación on-the-fly que permitirá gestionar **servicios web** de forma cómoda.

Versión 1.7: Otra versión cuyos cambios a nivel del lenguaje son limitados. Se produce una mejora de la máquina virtual incluyendo nuevos recolectores de basura.

Versión 1.8: Llega Java 8 (septiembre del 2014), el gran salto en cuanto al lenguaje se refiere. Se abren las puerta a la programación funcional con el uso de **expresiones Lambda y Streams**. Se realiza una revisión de APIS y se actualiza de forma importante la gestión de **fechas**.

Versión 1.9: Surge en septiembre del 2017 y añade mayor modularidad a la plataforma y permite la creación de nuevas arquitecturas.

A partir de la publicación de Java 9 se cambia el modelo de publicación de nuevas versiones optando por una basada en calendario en vez de una por características a incluir. El caso de versiones que han de incluir las características previstas ocasiona el problema que si una se retrasa provoca un retraso en la versión. Con el modelo basado en fechas fijas preestablecidas la versión se liberará con aquellas características que estén listas en la fecha planificada de publicación sin ser retrasadas por aquellas que no.

Versión 1.10: Surge en marzo del 2018 y Añade la inferencia de tipos.

Versión 1.11: Aparece en septiembre de 2018 y añade varias novedades importantes en cuanto a seguridad y elimina otras que en versio-

nes anteriores ya fueron marcadas como desaconsejadas.

La versión 1.12 aparecerá en marzo del 2019.

7.2. Interfaces funcionales

Se le conoce como interfaz funcional a toda aquella interfaz que tenga solamente **un método abstracto**.Puede implementar uno o más métodos de otros tipos, pero deberá tener forzosamente un único método sin implementar.

Las interfaces funcionales son un nuevo concepto agregado a partir de la versión 8 de Java y cobran su importancia al utilizar expresiones lambda.

7.2.1. Interfaces

Las **clases** en Java son plantillas para la creación de objetos.

```
1  public class Persona {
2
3      // Atributos
4      private String nombre;
5      private String primerApellido;
6      private String segundoApellido;
7      private int edad;
8
9      // Constructores
10     public Persona() {
11     }
12
13     public Persona(String nombre, String primerApellido, String
           segundoApellido, int edad) {
14         this.nombre = nombre;
15         this.primerApellido = primerApellido;
16         this.segundoApellido = segundoApellido;
17         this.edad = edad;
18     }
19
20     // Métodos
21     public String getNombre() {
22         return nombre;
23     }
24
25     public void setNombre(String nombre) {
26         this.nombre = nombre;
27     }
28
29     public String getPrimerApellido() {
30         return primerApellido;
31     }
32
```

```
33      public void setPrimerApellido(String primerApellido) {
34          this.primerApellido = primerApellido;
35      }
36
37      public String getSegundoApellido() {
38          return segundoApellido;
39      }
40
41      public void setSegundoApellido(String segundoApellido) {
42          this.segundoApellido = segundoApellido;
43      }
44
45      public int getEdad() {
46          return edad;
47      }
48
49      public void setEdad(int edad) {
50          this.edad = edad;
51      }
52      public void cumplirAños(){
53          this.edad += 1;
54      }
55  }
```

```
1           Persona p = new Persona();
2           p.setNombre("Pepe");
3           p.setEdad(56);
4           p.cumplirAños();
5           System.out.println(p.getEdad()); // 57
```

Las clases **públicas** (public class) son muy comunes. Son accesibles desde cualquier otra clase en la misma librería. Si están en otra libreria hay que importarlas (import javax.swing.JOptionPane;).

También hay clases **abstractas**. Son aquellas que tienen por lo menos un método abstracto. No implementan sus métodos, sino que dan las bases para que sean implementados.

De una clase abstracta no se tiene intención de crear objetos, sino que únicamente sirve para unificar datos y operaciones de subclases.

Las clases abstractas suelen contener métodos abstractos. Para que un método se considere abstracto ha de incluir en su declaración la palabra clave abstract. Además un método abstracto tiene estas peculiaridades:

a) No tiene cuerpo (llaves).

b) Su declaración termina con un punto y coma.

c) Sólo puede existir dentro de una clase abstracta. Dicho de otra manera, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.

d) Los métodos abstractos forzosamente habrán de estar sobreescritos en las subclases.

```
1  public abstract class Profesor extends Persona {
2
3      private String IdProfesor;
4
5      // Constructores
6      public Profesor ()
7      {
8        super();
9        IdProfesor = "Unknown";
10     }
11
12     public Profesor (String nombre, String apellidos, int edad,
           String id) {
13      super(nombre, apellidos, edad);
14      IdProfesor = id;
15    }
16
17     // Métodos
18
19     public void setIdProfesor (String IdProfesor) {
20        this.IdProfesor = IdProfesor;
21     }
22
23     public String getIdProfesor () {
24        return IdProfesor;
25    }
26
27     public void mostrarDatos() {
28         System.out.println ("Datos Profesor. Profesor de nombre: " +
               getNombre() + " " +
29         getApellidos() + " con Id de profesor: " + getIdProfesor() );
30    }
31
32     public String toString () {
33       return super.toString().concat(" -IdProfesor:
             ").concat(IdProfesor);
34     }
35
36     abstract public float importeNomina ();  // Método abstracto
37
38 }
```

```
1  public class ProfesorTitular extends Profesor {
2
3          public ProfesorTitular(String nombre, String apellidos, int
               edad, String id) {
4            super(nombre, apellidos, edad, id);
5          }
6
7          //Método abstracto sobreescrito en esta clase
8          public float importeNomina () {
9            return 30f * 43.20f;
10         }
11
12 }
```

Una **interfaz** en Java es una colección de métodos abstractos y propiedades constantes. No es una clase.

En las interfaces se especifica qué se debe hacer pero no su implementación, es decir, el cómo. Serán las clases que implementen estas interfaces las que describan la lógica del comportamiento de los métodos.

La principal diferencia entre interface y abstract es que un interface proporciona un mecanismo de encapsulación de los métodos sin forzar al usuario a utilizar la herencia.

El uso de las interfaces Java proporciona las siguientes ventajas:

- Organizar la programación.
- Permiten declarar constantes que van a estar disponibles para todas las clases que implementen esa interfaz.
- Obligar a que ciertas clases utilicen los mismos métodos (nombres y parámetros).

```
1
2  // Comparator es un de los muchos interfaz que java nos proporciona
3
4  // https://www.apuntesdejava.com/2009/04/comparable-y-comparator.html
5
6
7  import java.util.Comparator;
8
9  public class OrdenarPersonaPorAltura implements Comparator<Persona> {
10     @Override
11     public int compare(Persona o1, Persona o2) {
12       return o1.getAltura() - o2.getAltura(); // Devuelve un entero
             positivo si la altura de o1 es mayor que la de o2
13     }
14 }
```

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3
4  public class Programa {
5
6      public static void main(String arg[]) {
7
8          ArrayList<Persona> listaPersonas = new ArrayList<>();
9          listaPersonas.add(new Persona(1,"Maria",185));
10         listaPersonas.add(new Persona(2,"Carla",190));
11         listaPersonas.add(new Persona(3,"Yovana",170));
12         Collections.sort(listaPersonas, new
               OrdenarPersonaPorAltura());
13         System.out.println("Personas Ordenadas por orden total:
               "+listaPersonas);
14
15       }
```

```
16
17 }
```

Otro ejemplo:

```
1  ArrayList<String> lista = new ArrayList();
2         lista.add("Nieves");
3         lista.add("Ana");
4         lista.add("Teresa");
5
6         System.out.println("Lista sin ordenar");
7         visualizar(lista);
8
9
10        // Ordena de menor a mayor
11        lista.sort(null);
12        System.out.println("\nLista ordenada de menor a mayor");
13        visualizar(lista);
14
15
16        // Quiero ordenar de mayor a menor usando el interfaz
              Comparator
17        // Lo uso directamente sin crear la clase que lo implementa.
18        lista.sort(new Comparator<String> (){
19            @Override
20            public int compare(String n1, String n2)
21            {
22                return n2.compareTo(n1);
23            }
24        });
25        System.out.println("\nLista ordenada de mayor a menor");
26        visualizar(lista);
27
28
29    }
30    public static void visualizar(ArrayList<String> lista)
31    {
32        for(String l: lista)
33            System.out.println(l);
34
35        lista.stream().forEach((l) -> {
36                        System.out.println(l);
37                       });
38    }
39 }
```

Una clase puede implementar varios interfaces.

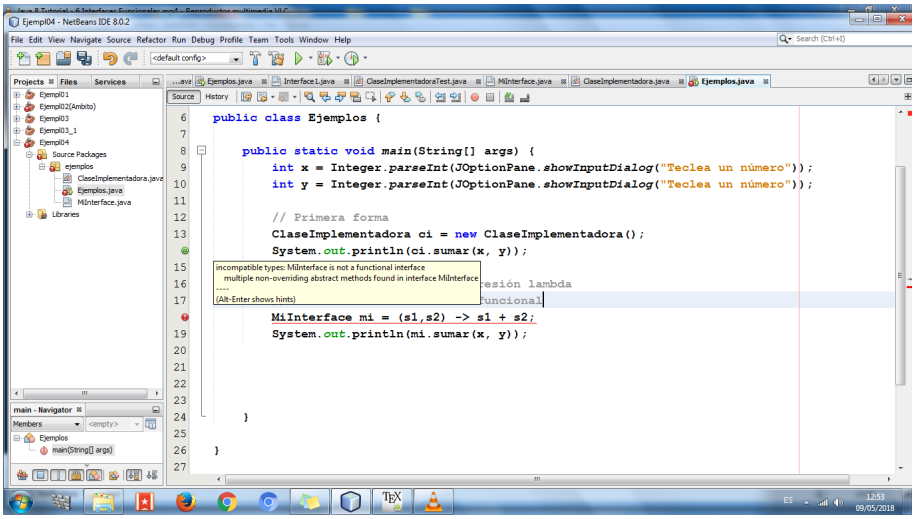
Dentro de los interfaces podemos crear **métodos por defecto**, métodos con código que podrán ser utilizado en todas las clases que implementen el interfaz.

7.2.2. Interfaz funcional

Como ya hemos comentado antes, un interfaz funcional es toda interfaz que tenga solamente **un método abstracto**.Puede implementar uno o más métodos de otros tipos, pero deberá tener forzosamente un único método sin implementar.

Las interfaces funcionales son un nuevo concepto agregado a partir de la versión 8 de Java y cobran su importancia al utilizar expresiones lambda.

```
1  //ordenar de mayor a menor usando el interfaz funcional Comparator
2
3          lista.sort((String n1, String n2) -> n2.compareTo(n1));
4          System.out.println("\nLista ordenada de mayor a menor");
5          visualizar(lista);
```



Para que por error no nos permita poner más de un método existe la anotación FunctionalInterface.

```
1  package ejemplos;
2
3  @FunctionalInterface
4  public interface MiInterface {
5      // Sólo un método sin código --- Interface funcional
6      public int sumar(int x, int y);
7      //public int restar(int x, int y);
8
9  }
```

7.3. Funciones anónimas o lambda

Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan nombre. Su sintaxis básica es:

El **operador** lambda (la flecha) que separa la declaración de parámetros de la declaración del cuerpo de la función.

Parámetros:

Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.

Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.

Cuerpo de la función lambda:

Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no se necesita especificar la clausula return en el caso de que deban devolver valores.

Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la clausula return en el caso de que la función deba devolver un valor .

```
() -> expresión

parametro -> expresión

(parametros) -> expresión

(parametros) -> { sentencias; }
```

```
public static void main(String[] args) {
        // Función que crea un arrayList.
        // No recibe ningún parametro.
        // Si hay return.
        ArrayList<String> lista = crearArrayList();

        //() -> new ArrayList<String>()


        // Función que suma dos números que se le pasa como
            parámetros.
        // Hay parámetros y hay return
        int resultado = sumar(3,5);

        //(int a, int b) -> a+b


        // Función que devuelve true si un número es mayor que 10.
        // Hay parámetro, hay return y más de una línea.
        boolean mayor = comparar(15);

```

```
21          /*(int a) -> {
22                if (a > 10)
23                    return true;
24                else
25                    return false;
26        }*/
27
28
29          // Función que recibe un dato y lo muestra.
30          // Hay parámetro y no return.
31          mostrar("Hola");
32
33          //(elemento)-> System.out.println(elemento)
34
35
36          // Función que suma dos a un número.
37          // Hay parámetro y return.
38          resultado = sumar2(4);
39
40          //int z -> z + 2;
41
42
43          // Función que muestra un mensaje (no hay ni parámetro ni
                return)
44          mostrar();
45
46          //() -> System.out.println(" Mensaje 1 ")
47
48      }
49
50      public static ArrayList<String> crearArrayList(){
51          return new ArrayList<String>();
52      }
53
54      public static int sumar(int a, int b){
55          return a + b;
56      }
57
58      public static boolean comparar(int a){
59          if (a > 10)
60              return true;
61          else
62              return false;
63      }
64
65      public static void mostrar(String dato){
66          System.out.println(dato);
67      }
68
69      public static int sumar2(int z){
70          return z + 2;
71      }
72
73      public static void mostrar(){
74          System.out.println("Mensaje 1");
75      }
```

Las lambdas **pueden ser utilizadas allá donde el tipo de paráme-**

tros aceptados sea una interfaz funcional. En vez de implementar el método, se codifica directamente.

```
1  package ejemplos;
2
3  public interface MiInterface {
4      // Sólo un método sin código --- Interface funcional
5      public int sumar(int x, int y);
6
7  }
```

```
1  package ejemplos;
2
3  public class ClaseImplementadora implements MiInterface{
4
5      // Código generado automáticamente para solucionar el error que
           da
6     /* @Override
7      public int sumar(int x, int y) {
8          throw new UnsupportedOperationException("Not supported
               yet."); //To change body of generated methods, choose
               Tools | Templates.
9      }*/
10
11
12     // Cambio el método y lo codifico adecuadamente
13     @Override
14     public int sumar(int x, int y) {
15         return x + y;
16     }
17
18 }
```

```
1  package ejemplos;
2
3  import javax.swing.JOptionPane;
4
5  public class Ejemplos {
6
7      public static void main(String[] args) {
8          int x = Integer.parseInt(JOptionPane.showInputDialog("Teclea
               un número"));
9          int y = Integer.parseInt(JOptionPane.showInputDialog("Teclea
               un número"));
10
11         // Primera forma
12         ClaseImplementadora ci = new ClaseImplementadora();
13         System.out.println(ci.sumar(x, y));
14
15         // Segunda forma con una expresión lambda
16         MiInterface mi = (s1,s2) -> s1 + s2;
17         System.out.println(mi.sumar(x, y));
18     }
```

7.3.1. Referencias a métodos

Las referencias a los métodos nos permiten reutilizar un método como expresión lambda. Para hacer uso de las referencias a métodos basta con utilizar la siguiente sintaxis:

referenciaObjetivo::nombreDelMetodo.

Con las referencias a los métodos se ofrece una anotación más rápida para expresiones lambda simples y existen tipos diferentes:

- **A métodos estáticos**. Estos métodos no pueden tener parámetros ya que Java todavía no implementa la currificación.

```
1  package ejemplos;
2
3  public interface MiInterface {
4      public void metodo();
5  }
```

```
1  package ejemplos;
2
3  import javax.swing.JOptionPane;
4
5  public class Ejemplos {
6      static int x,y;
7      public static void visualizarSuma(){
8          int numero = x + y;
9          System.out.println("El resultado es: " + numero);
10     }
11
12     public static void main(String[] args) {
13         x =
               Integer.parseInt(JOptionPane.showInputDialog("Teclea
               un número"));
14         y =
               Integer.parseInt(JOptionPane.showInputDialog("Teclea
               un número"));
15
16         // Implementación del método de la interfaz funcional
17         // El método realiza una llamada a un método estático
18
19         // Primera forma
20         //MiInterface mi = () -> Ejemplos.visualizarSuma();
21
22         // Segunda forma:: Operador de referencias a métodos
23         MiInterface mi = Ejemplos::visualizarSuma;
24
25         // Llamada al método.
26         mi.metodo();
27     }
28 }
```

- **A métodos instanciados.**

```java
package ejemplos;

public class Persona {
    String nombre;
    int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public void cumpleaños(){
        this.edad = this.edad + 1;
    }
}
```

```java
package ejemplos;

public interface MiInterface {
    public void metodo(Persona[] listaPersonas);
}
```

```java
public class Ejemplos {

    public void sumar(Persona [] lista)
    {
       for(Persona p:lista)
           p.cumpleaños();
    }

    public static void main(String[] args) {
        // Instancio la clase Ejemplos para que no sea un
            método estático
        Ejemplos ejemplo = new Ejemplos();

        Persona[] listaPersonas = {new Persona("Pepe",45),new
            Persona("Juan",54),new Persona("Manolo",28)};

        //MiInterface mi = (lista) -> ejemplo.sumar(lista);
        MiInterface mi = ejemplo::sumar;
        mi.metodo(listaPersonas);
```

```
18
19          for(Persona p:listaPersonas)
20              System.out.println(p.getNombre() + p.getEdad());
21      }
22
23  }
```

- **Al método constructor.**

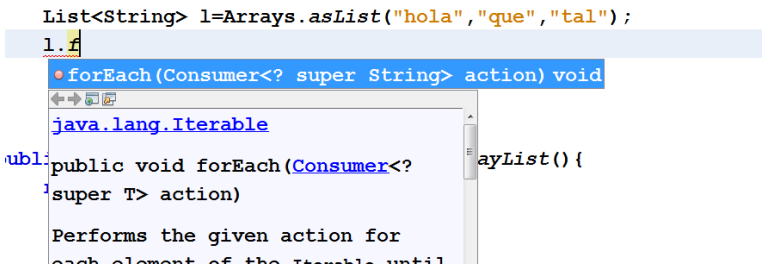
```
1  package ejemplos;
2
3  public interface MiInterface {
4      public Persona crearPersona(String nombre, int edad);
5  }
```

```
1  package ejemplos;
2
3  public class Ejemplos {
4
5      public static void main(String[] args) {
6
7          /* MiInterface mi = new MiInterface(){
8
9              @Override
10             public Persona crearPersona(String nombre, int
                   edad) {
11                 return new Persona(nombre,edad);
12             }
13         };*/
14
15         // MiInterface mi = (String nombre,int edad)->new
               Persona(nombre,edad);
16
17         MiInterface mi = Persona::new;
18
19         Persona p1 = mi.crearPersona("Pepe", 80);
20         Persona p2 = mi.crearPersona("Juan", 20);
21
22         System.out.println(p1.getNombre() + p1.getEdad());
23         System.out.println(p2.getNombre() + p2.getEdad());
24     }
25
26 }
```

7.3.2. Tipos de interfaces funcionales

https://www.arquitecturajava.com/java-8-functional-interfaces-y-sus-tipos/

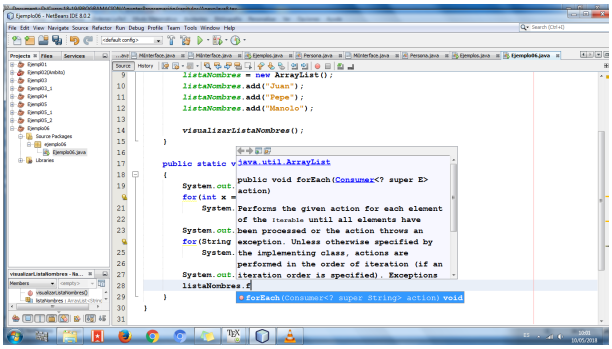
- **Consumers**: Función que reciba parámetros pero que no devuelva nada.
- **Functions**: reciben uno o más parámetros y retornan un resultado.



- **Suppliers**: función que no recibe parámetro alguna y devuelve un resultado.
- **Predicates**: Los interfaces de tipo predicado son una especialización de los Functions ya que reciben un parámetro y devuelven un valor booleano.
- **Operators**: Se trata de interfaces funcionales que reciben un tipo de parámetro y devuelven un resultado que es del mismo tipo.

7.4. Colecciones

7.4.1. forEach



El método forEach pide un parámetro Consumer que es un interfaz funcional con un único método llamado accept , que nos permite utilizar funciones lambda.

```
1  package ejemplo06;
2
3  import java.util.ArrayList;
4
5  public class Ejemplo06 {
6
7      static ArrayList<String> listaNombres;
8
9      public static void main(String[] args) {
10         listaNombres = new ArrayList();
11         listaNombres.add("Juan");
12         listaNombres.add("Pepe");
13         listaNombres.add("Manolo");
14
15         visualizarListaNombres();
16     }
17
18     public static void visualizarListaNombres()
19     {
20         System.out.println("For tradicional");
21         for(int x = 0; x < listaNombres.size(); x++)
22             System.out.println(listaNombres.get(x));
23
24         System.out.println("\nFor mejorado");
25         for(String nombre:listaNombres)
26             System.out.println(nombre);
27
28         System.out.println("\nForEach");
29         // Expresión lambda
30         listaNombres.forEach((nombre)->System.out.println(nombre));
31
32         // Con referencias a métodos (método println)
```

```
33          listaNombres.forEach(System.out::println);
34
35          listaNombres.stream().forEach(System.out::println);
36      }
37  }
```

En vez de utilizar directamente el método forEach de las listas se puede hacer uso del forEach de la clase Stream. De esta manera conseguimos acceso a los método map, filter, ... que se verán más adelante

7.4.2. removeIf

El método removeIf nos permite eliminar elementos de una colección utilizando expresiones lambda.

```
1   package ejemplo06;
2
3   import java.util.ArrayList;
4   import java.util.Iterator;
5
6   public class Ejemplo06 {
7
8       static ArrayList<String> listaNombres;
9
10      public static void main(String[] args) {
11          listaNombres = new ArrayList();
12          listaNombres.add("Juan");
13          listaNombres.add("Pepe");
14          listaNombres.add("Manolo");
15          listaNombres.add("Juan");
16          listaNombres.add("Ana");
17          listaNombres.add("Pepe");
18
19          borrarNombres();
20      }
21
22      public static void borrarNombres()
23      {
24          // Quiero borrar a Pepe
25          // Métodos "tradicionales"
26          for(int x = 0; x < listaNombres.size(); x++)
27             if (listaNombres.get(x).compareTo("Pepe")==0)
28                 listaNombres.remove(x);
29          listaNombres.forEach(System.out::println);
30
31          while (listaNombres.contains("Pepe"))
32              listaNombres.remove("Pepe");
33           listaNombres.forEach(System.out::println);
34
35           Iterator<String> it= listaNombres.iterator();
36           while (it.hasNext())
37           {
38                  if (it.next().equals("Pepe")) {
39                  it.remove();
40           }
41
42          listaNombres.removeIf((nombre)->nombre.equalsIgnoreCase("Pepe"));
43          // compareTo no sirve. Necesitamos un valor booleano.
44          listaNombres.forEach(System.out::println);
45      }
```

```
46 }
```

7.4.3. sort

```
1  public static void ordenarNombres()
2     {
3          // Ordena de menor a mayor
4          listaNombres.sort(null);
5          listaNombres.forEach(System.out::println);
6
7          // Ordeno de mayor a menor
8          listaNombres.sort((nombre1,
               nombre2)->-nombre1.compareTo(nombre2));
9          listaNombres.forEach(System.out::println);
10    }
```

7.5. Stream

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

http://www.oracle.com/technetwork/es/articles/java/procesamiento-streams-java-s html

java.util.stream es un nuevo API (Application Programming Interface) que aparece con Java 8 que permite la programación funcional sobre un flujo de valores sin estructura.

Stream se define como una secuencia de elementos que provienen de una fuente que soporta operaciones para el procesamiento de sus datos de forma declarativa usando expresiones lambda; permitiendo el posible encadenamiento de varias operaciones, con lo que se logra tener un código fácil de leer y con un objetivo claro.

Esta API nos permite realizar operaciones sobre colecciones de datos usando el modelo filtro/mapeo/reducción, en el cual se seleccionan los datos que se van a procesar (filtro), se convierten a otro tipo de dato (mapeo) y al final se obtiene el resultado deseado (reducción).

Un filtro (**filter**) se utiliza para quedarse sólo con los elementos que cumplan ciertas condiciones.

```
1  public static void main(String[] args) {
2          ArrayList<String> listaNombres = new ArrayList();
3          listaNombres.add("Aitor");
4          listaNombres.add("Iñigo");
5          listaNombres.add("Sergio");
6          listaNombres.add("Jon");
7          listaNombres.add("Mikel");
8          listaNombres.add("Aitor");
9          listaNombres.add("Ander");
10
11         /* Queremos quedarnos solo con los nombres que
12         comienzan con la letra A*/
13
14         // Con cualquier repetitiva
15
16         // Con removeIf
17         listaNombres.removeIf((n)->!n.startsWith("A"));
18         listaNombres.forEach(System.out::println);
19
20         // Con filter
21         listaNombres.stream().filter((n)->n.startsWith("A")).forEach(System.out::println);
22
23     }
```

Con el método **sorted** podemos ordenar los elementos indicando, si es necesario, como queremos hacerlo.

```
1  public static void main(String[] args) {
2          ArrayList<String> listaNombres = new ArrayList();
3          listaNombres.add("Aitor");
4          listaNombres.add("Iñigo");
5          listaNombres.add("Sergio");
6          listaNombres.add("Jon");
7          listaNombres.add("Mikel");
8          listaNombres.add("Aitor");
9          listaNombres.add("Ander");
10
11         // Queremos ordenar alfabeticamente
12         listaNombres.stream().sorted().forEach(System.out::println);
13
14         // De manera descendente
15         listaNombres.stream().sorted((n1,n2)->
               n2.compareTo(n1)).forEach(System.out::println);
16
17     }
```

Con el método **map** podremos transformar cada uno de los elementos de la colección con la operación indicada.

```
1   public static void main(String[] args) {
2          ArrayList<String> listaNombres = new ArrayList();
3          listaNombres.add("Aitor");
4          listaNombres.add("Iñigo");
5          listaNombres.add("Sergio");
6          listaNombres.add("Jon");
7          listaNombres.add("Mikel");
8          listaNombres.add("Aitor");
9          listaNombres.add("Ander");
10
11         // Paso a mayúsculas
12         listaNombres.stream().map((n)->n.toUpperCase()).forEach(System.out::println);
13         listaNombres.stream().map(String::toUpperCase).forEach(System.out::println);
14
15     }
```

Hay más métodos dentro de stream tales como limit, count, reduce, etc....

7.6. Java 9

https://picodotdev.github.io/blog-bitix/2017/09/novedades-de-java-9-mas-alla-d

https://www.arquitecturajava.com/java-9-collections-y-sus-novedades/

https://www.adictosaltrabajo.com/tutoriales/modularidad-en-java-9-12/

https://www.adictosaltrabajo.com/2017/10/30/modularidad-en-java-9-12/

La incorporación de los **módulos** a la plataforma con Java 9 es una de las modificaciones más importantes en esta nueva versión.

Los módulos van a mejorar una de las deficiencias existentes en la visibilidad de las clases entre paquetes. Los módulos de Java proporcionan una mayor encapsulación de las clases contenidas en un paquete y las librerías. Esta encapsulación evita que una aplicación u otra librería haga uso y dependa de clases y paquetes de los que no debería lo que mejora la compatibilidad con versiones futuras.

7.7. Novedades Java 10

https://picodotdev.github.io/blog-bitix/2018/03/novedades-de-java-10/

https://www.lomasnuevo.net/noticias/lo-nuevo-de-java-10/

El 20 de marzo de 2018 se publicó la que es la versión 10 de Java siguiendo el nuevo calendario de publicar una nueva versión cada seis meses. Java 10 tiene una lista más reducida de cambios que Java 9 pero importantes y significativos. Java es el último en unirse a la fiesta de la inferencia de tipos pero ha sido de forma intencionada ya que el coste de implementarla de forma incorrecta supone un alto coste que hay que mantener en adelante.

- (var) Inferencia del tipo de variable local, para mejorar el lenguaje Java y extender la inferencia del tipo a declaraciones de variables locales con inicializadores.
  https://translate.google.com/translate?depth=1&hl=es&prev=search&rurl=translate.google.es&sl=en&sp=nmt4&u=https://medium.com/%40afinlay/java-10-sneak-peek-local-variable-type-inference-v&xid=17259,15700021,15700043,15700124,15700149,15700168,15700173,15700186,15700189,15700201
  Restricciones:
  - Sólo se puede usar var con:
    1. Variables locales con inicializadores. Solo var x; es un error.
    2. Índices de bucle forzado
    3. Dentro del alcance local de un bucle condicional.
  - No se pueden instanciar múltiples variables.

```
1
2   var x = 0, y = 0;  // ¡no!
```

  - Sin Lambdas, referencias de método o inicializadores de matriz.

```
1
2 var l = ()->{}; // ¡no!
3
4 var m = java.lang.Math::pow // ¡no!
5
6 var a = {2,3,4} // ¡no!
```

  - Sin valores nulos.

```
1
2   var x = null;  //¡No!
```

- No se puede reasignar a un tipo diferente

```
1
2    var x = 0;  // Inferido para ser del tipo 'int'
3
4    var x = "c";  //¡No!  - ¡tipos incompatibles!
```

```
1
2  var nombre = "Pepe";
3
4  System.out.println(nombre.getClass());
5  // class java.lang.String
6
7  var edad =6;
8  // Convierto a Object. Si me quedo con int no hay clase.
9  System.out.println(((Object)edad).getClass());
10 // class java.lang.Integer
```

7.8. Java 11

https://www.campusmvp.es/recursos/post/java-11-ya-esta-aqui-te-toca-pagar-a-or
aspx

https://picodotdev.github.io/blog-bitix/2018/09/novedades-y-nuevas-caracteristi

El JDK 11 inicia una nueva era en la licencia de uso. Hasta ahora podías descargar y programar con el Kit de Desarrollo de Java oficial de Oracle y luego poner tu aplicación en producción o distribuirla sin tener que pagar nada al gigante del software. Sin embargo, a partir de Java 11 y del JDK 11, aunque puedes seguir desarrollando con él, tendrás que pagar una licencia a Oracle si quieres utilizarlo para poner las aplicaciones en producción. El coste es de 2,5 dólares al mes por cada usuario de escritorio, y de 25 dólares por procesador en el caso de aplicaciones de servidor.

Esto no afecta a versiones anteriores del JDK, por lo que si usas Java 8, 9 o 10 sigue siendo gratuito.

Las novedades de esta nueva versión no son muy significativas. Entre ellas , por ejemplo, está:

- Los parámetros de una lambda pueden declararse con var. Esta funcionalidad tiene algunas restricciones. No se puede mezclar el uso y no uso de var y no se puede mezclar el uso de var y tipos en lambdas explícitas. Son consideradas ilegales por el compilador y producirá un error en tiempo de compilación.

```
1
2  (x, y) -> x.process.(y)
3
4  (var x, var y) -> x.process.(y)
5
6  (@NotNull var x, @NotNull var y) -> x.process.(y)
7
8  (var x, y) -> x.process.(y)
9  // no se puede mezclar var y no var
10
11 (var x, int y) -> x.process.(y)
12 // no se puede mezclar var y tipos en lamdas explícitas
```