Sumario

Paquetes principales, clases principales y Expresiones

Paquetes importantes de Java

Estos son los paquetes más importantes de la API de Java:

Paquete	Descripción
`java.applet`	Contiene clases para la creación de applets.
`java.awt`	Contiene clases para crear interfaces de usuario con ventanas.
`Java.date`	Contiene clases de manejo de fecha.
`java.io`	Contiene clases para manejar la entrada/salida.
`java.lang`	Contiene clases variadas pero imprescindibles para el lenguaje, como *Object*, *Thread*, *Math*... El paquete `java.lang` es importado por defecto en el ficheros de código Java.
`java.net`	Contiene clases para soportar aplicaciones que acceden a redes TCP/IP.
`java.nio`	Contiene clases para el manejo de Archivos
`java.time`	Contiene clases de manejo de tiempo a partir de JDK 8
`java.util`	Contiene clases que permiten el acceso a recursos del sistema, Collections, etc.
`java.sql`	Contiene clases de interface SQL.
`javafx`	Contiene la interface gráfica fx, no se sabe si seguirá incluida en el JDK o sera un modulo a parte
`javax.swing`	Contiene clases para crear interfaces de usuario mejorando la AWT.

Paquete jdk.incubator

Este paquete se incorporo en java 9 y contienen módulos experimentales que pueden ser eliminados o cambiados en próximas versiones del JDK. Normalmente se agregan cosas en este módulo en veriones no LTS y se presenta la versión final en la versión LTS.

El ejemplo mas típico es el del cliente hhtp2 que salio en la versión jdk10, en el paquete `jdk.incubator.httpclient` y que fue modificado el api y cambiado a java.net para la versión jdk11.

En software de producción no es recomendado usar clases de este paquete por que pueden alterados en el proceso de evolución del lenguaje.

Hay muy poca documentación al respecto se puede consultar http://openjdk.java.net/jeps/11

Clases Importantes

La clase System

Definición

La clase System representa al sistema donde se esta ejecutando el programa Java. Por lo tanto se la puede utilizar para interactuar con el entorno en el que corre, y utilizar las propiedades del entorno, sistema operativo, usuario y demás.

Dónde utilizarla

Puede accederse a la salida estandard del proceso, a la salida de error del proceso y a la entrada estandard del proceso mediante, System.out, System.err, System.in respectivamente, de modo de poder interactuar con el Sistema Operativo. También permite obtener el tiempo en milisegundos llamando al sistema operativo para lograr este objetivo.
Otra cosa muy útil de la Clase System es que permite cargar librerías externas en una linea de ejecucion con el metodo load(), mediante esta operación se logra cargar librerías en tiempo de corrida del programa y sumar así funcionalidades a nuestros programas.
Posee atributos y métodos de uso general, y son todos estáticos, es decir atributos y métodos de clase. No es una clase instanciable.

Entre los métodos mas conocidos estan:

```
.exit():          Termina la ejecución de la Java Virtual Machine.
.gc():            Invoca al Garbage Collector.
.getProperties(): Trae todas las propiedades del sistema.
.getProperty():   Trae una propiedad en particular del sistema.
.getenv():        Trae un mapa de propiedades de entorno.
```

Propiedades del sistema

Como la máquina virtual genera su propio espacio de ejecución para las aplicaciones, existen una serie de valores de entorno para dicho ambiente. La forma de sustituir los valores de entorno que se pueden definir para una aplicación en un sistema operativo, Java las maneja como una serie de propiedades que pueden ser consultadas.
A estas propiedades se las denomina propiedades del sistema y es un concepto que se utiliza para remplazar las variables de entorno (las cuales son específicas de la plataforma que las define).
Como todo en Java, los valores se almacenarán en un objeto, el cual es del tipo Properties . Este objeto lo define la máquina virtual al arrancar y se puede obtener una referencia al mismo por el método estático System.getProperties() , el cual retorna un objeto de este tipo con los valores antes mencionados.
Una vez obtenida la referencia al objeto que almacena los valores, se puede utilizar un método de servicio, el cual está definido en este, para recuperar el valor asociado al nombre de una propiedad que le sea especificado como argumento. El método getProperty() retorna un String el cual representa el valor de la propiedad cuyo nombre se especifica.

```
//Recorrido del mapa de propiedades
System.getProperties().forEach((k,v)→System.out.println(k+” : “+v);

//Pedido de una propiedad en particular
System.getProperty(“user.name”);

//Recorrido del mapa de propiedades de entorno
System.getenv().forEach((k,v)->System.out.println(k+": "+v));
//Pedido de una propiedad en particular
```

```
System.out.println(System.getenv("PATH"));
```

Hay que tener en cuenta que dependiendo la configuración de cada sistema, esta salida será diferente.

La clase Properties implementa un mapa de nombres respecto de sus valores (un mapa de String a String) donde se debe interpretar el primer String como una clave y el segundo como el valor asociado a dicha clave.
El método propertyNames() retorna una referencia a un objeto del tipo Enumeration (enumeración) de todos los nombres de propiedades. Esta sirve para recorrer las propiedades y obtener sus valores.
El método getProperty() retorna un String representando el valor que corresponde al nombre indicado en el argumento del método.

Método .exit()

Este método termina la ejecución de la Java Virtual Machine (cierra el programa). Se debe colocar un parámetro entero que sera la devolución al sistema operativo.

```
System.exit(0) or EXIT_SUCCESS;  ---> Success
System.exit(1) or EXIT_FAILURE;  ---> Exception
System.exit(-1) or EXIT_ERROR;   ---> Error
```

Salidas por consola

Los sistemas operativos modernos definen un mínimo de tres corrientes de caracteres estándar para el acceso a los dispositivos periféricos:

- El ingreso estándar (Standard Input) System.in
- La salida estándar (Standard Output) System.out
- La corriente de errores estándar (Standard Error) System.err

Las corrientes estándar están asociadas siempre a un dispositivo por defecto. Sin embargo. Como son corrientes (en inglés, streams) pueden ser dirigidas hacia a otros dispositivos (como el ejemplo típico de la salida por pantalla que se direcciona a la impresora).

El estándar output

Java define una serie de clases que permiten un cómodo manejo unificado de todas las corrientes en un sistema operativo (entre las que se encuentran las estándar y las propias de los archivos en disco, canales de comunicación, etc...).
Para llevar a cabo esa unificación, se definen una serie de clases que reciben como parámetros a las corrientes, por ejemplo, y definen métodos en su interfaz que permiten un mejor manejo de las mismas. Esta funcionalidad a nivel de diseño se la denomina “decorador” y es un patrón de diseño conocido.
Así, a través de los decoradores, se pueden utilizar métodos conocidos para escribir o leer de una corriente (los detalles de esta operatoria se explicarán posteriormente). Se puede definir entonces la funcionalidad de algunos métodos conocidos que permiten interaccionar con las corrientes:

- Los métodos println escriben el argumento y una nueva línea.
- Los métodos print imprimen el argumento sin la nueva línea

Ambos métodos están sobrecargados para todos los tipos primitivos y además para char[] , Object

y String .
Los métodos print(Object) y println(Object) llaman al método toString() del argumento para permitir al programador que coloque de la forma que quiera un mensaje que identifique al objeto tan sólo sobrescribiendo toString .

El estándar input

Para demostrar la interacción con la corriente estándar, el siguiente código lee los ingresos por teclado hasta que se cierra la corriente con un carácter de fin de archivo.

```
String s;
// Crea un lector con buffer para cada línea del teclado.
InputStreamReader ir = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(ir);
System.out.println("Unix: Tipear ctrl-d o ctrl-c para salir."
      + "\nWindows: Tipear ctrl-z para salir ");
try {
      // Lee cada línea de entrada y lo muestra por pantalla.
      s = in.readLine();
      while (s != null) {
            System.out.println("Leído: " + s);
            s = in.readLine();
      }
      // Cierra el lector con buffer
      in.close();
} catch (IOException e) {
      e.printStackTrace();
}
```

La clase Date y Calendar

Desde la versión Java 1.0, los años en el API empiezan en 1900 y los meses inician con el índice 0. Por lo tanto, cualquier fecha representada haciendo uso de la clase java.util.Date no era legible. Por ejemplo, representaremos la fecha en que Java 8 fue liberado 18 de Marzo del 2014:

```
Date date = new Date(114, 2, 18);
System.out.println(date);
```

Como resultado de la línea anterior tenemos lo siguiente:

```
Tue Mar 18 00:00:00 AEST 2014
```

En Java 1.1, la clase java.util.Calendar fue agregada y varios métodos de la clase java.util.Date fueron deprecados. Hacer uso de la clase java.util.Calendar no dio mayores beneficios dado su diseño.

```
Calendar calendar = new GregorianCalendar(2014, 2, 18);
System.out.println(calendar.getTime());
```

Month es una enumeración. Recuerde que una enumeración no es un int y no puede compararse con uno,

por ejemplo:

```
Month month = Month.JANUARY;
//boolean b1 = month == 1;          // DOES NOT COMPILE
```

```
boolean b2 = month == Month.APRIL;  // false
```

La clase LocalDateTime

A partir de Java 8, la nueva Date API es amigable y esta basada en el proyecto Joda-Time la cual fue adoptada como JSR-310.
En el paquete java.time, se puede encontrar las nuevas clases del Date API como LocalDate, LocalTime, LocalDateTime, Instant, Duration y Period, de las cuales hablaremos a continuación.

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
```

LocalDateTime: Contiene la fecha y la hora, pero no contiene la zona horaria
LocalTime: Contiene la hora, no contiene la fecha ni la zona horaria.
LocalDate: Contiene la fecha, no contiene la hora ni la zona horaria.

Formateando Fecha y Hora

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
System.out.println(date.getDayOfWeek());       // MONDAY
System.out.println(date.getMonth());           // JANUARY
System.out.println(date.getYear());            // 2020
System.out.println(date.getDayOfYear());       // 20
```

```
//Como manipular LocalDate
LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
System.out.println(date);   // 2014-01-20

date = date.plusDays(2);
System.out.println(date);   // 2014-01-22

date = date.plusWeeks(1);
System.out.println(date);   // 2014-01-29

date = date.plusMonths(1);
System.out.println(date);   // 2014-02-28

date = date.plusYears(5);
System.out.println(date);   // 2019-02-28


LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(dateTime);           // 2020-01-20T05:15

dateTime = dateTime.minusDays(1);
System.out.println(dateTime);           // 2020-01-19T05:15

dateTime = dateTime.minusHours(10);
System.out.println(dateTime);           // 2020-01-18T19:15

dateTime = dateTime.minusSeconds(30);
System.out.println(dateTime);           // 2020-01-18T19:14:30


// Haciendo un poco de programación funcional.
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time)
```

```
            .minusDays(1).minusHours(10).minusSeconds(30);
```

La clase String

Los objetos String son secuencias inmutables de caracteres Unicode, esto quiere decir, una vez creado no se pueden alterar. Cualquier intento de modificarlo generará la creación de un nuevo objeto del tipo y el anterior será seleccionable para el recolector de basura (garbage collector).
Las operaciones que pertenecen a este tipo de objeto pueden crear nuevos Strings, como ser: concat, replace, substring, toLowerCase, toUpperCase y trim. Lo que obviamente se debe hacer es asignarlo a otra referencia del mismo tipo si no se quiere perder el que generó la invocación.

Posee operaciones de:
Búsqueda: endsWith, startsWith, indexOf y lastIndexOf, containt.
Comparaciones: equals, equalsIgnoreCase y compareTo.
Otras: charAt y length.

Inicialización de un String

Las formas de declarar e inicializar un objeto de la clase String son las siguientes:

```
String cadena = "Soy una cadena de caracteres";
String cadena = new String("Soy una cadena de caracteres");
char[] caracteres = {'h','o','l','a'};
String cadena = new String(caracteres);
```

Todos los objetos en java heredan de la clase Object, y la clase Object tiene un método toString() que nos retorna la representación del objeto en un String.
Este método puede ser sobreescrito de modo tal que la información que se retorne en el String resultado sea propia de la instancia de la clase.

```
//Algunas curiosidades de la clase String

String s1 = "bunny";
String s2 = "bunny";
String s3 = new String("bunny");
System.out.println(s1 == s2);            // true
System.out.println(s1 == s3);            // false
System.out.println(s1.equals(s3));       // true


String s4 = "1" + 2 + 3;
String s5 = 1 + 2 + "3";
System.out.println(s4); // 123
System.out.println(s5); // 33
```

Métodos más comunes de la clase String

```
String cadena="El que se fue a Sevilla perdio su silla y el que se fue al
                Torreon, su sillon";

        System.out.println(cadena.charAt(0)); // Nos devolverá E

        System.out.println(cadena.charAt(11)); //Nos devolverá u
```

```
  System.out.println(cadena.endsWith("n")); //Nos devuelve true

  System.out.println(cadena.startsWith("e")); //Nos devuelve false, Java
      distingue entre mayúsculas y minúsculas

  System.out.println(cadena.equals("El que se fue a Sevilla perdio su
      silla y el que se fue al Torreon, su sillon")); //Nos devuelve true

  System.out.println("Hola".equalsIgnoreCase("HOLA")); //Nos devuelve true

  byte[] array_bytes=cadena.getBytes(); //Creamos un array de bytes e
      insertamos la devolución del metodo

  System.out.println("Codigo ASCII de cada caracter");
  for (int i=0;i<array_bytes.length;i++){
      System.out.print(array_bytes[i]+" "); //Muestra los codigos
  }
  System.out.println("");
  System.out.println(cadena.indexOf("fue")); //localiza la posicion donde
      se encuentra "fue"

  System.out.println(cadena.length()); // Nos devuelve la longitud: 77

  System.out.println(cadena.replace('a', 'e')); //Cambia todas las a por e

  System.out.println(cadena.toLowerCase()); //Transforma el String a
      minúsculas

  System.out.println(cadena.toUpperCase()); //Transforma el String a
      mayúsculas

  System.out.println(cadena.trim()); //Elimina los espacios adelante y
      atrás de la cadena

  String[] vector=cadena.split(" "); //Genera un vector usando el caracter
      del parámetro como separador
  for(String s:vector) System.out.println(s);

  System.out.println(cadena.substring(10)); //Devuelve la cadena quitando
      los 10 primeros caracteres

  System.out.println(cadena.substring(10, 20)); //Devuelve parte de la
      cadena, desde el caracter indice 10 hasta el 20

  //String format inyecta parametros en la cadena %s para String y %d para
      números
  System.out.println(String.format("Hoy es %S, Hora %d", "Jueves",10)); //
      Jueves en Mayúsculas.
  System.out.println(String.format("Hoy es %s, Hora %d", "Jueves",10)); //
      Jueves queda igual.


String s = "abcde ";
System.out.println(s.trim().length());                    // 5
System.out.println(s.charAt(4));                          // e
System.out.println(s.indexOf('e'));                       // 4
System.out.println(s.indexOf("de"));                      // 3
System.out.println(s.substring(2, 4).toUpperCase());      // CD
System.out.println(s.replace('a', '1'));                  // 1bcde
System.out.println(s.contains("DE"));                     // false
System.out.println(s.startsWith("a"));                    // true
```

String compact jdk9

Internamente un String almacena un array de caracteres (unicode 2 bytes). A partir de java 9 hay un cambio importante llamado “Compact String” que almacena un array de bytes.
Hasta java 8 el String era almacenado como:

```
private final char[] value;
```

A partir de java 9 el String es almacenado como:

```
private final byte[] value;
```

Las clases StringBuffer y StringBuilder

Definición

Los objetos del tipo StringBuffer y StringBuilder almacenan en su interior secuencias de caracteres Unicode que pueden cambiar. Los objetos de estos tipos deben ser utilizados cada vez que se quiera trabajar con el formato de un String cuyo tamaño pueda ser variable. La principal diferencia entre ambas clases es que StringBuilder es segura respecto de los threads, es decir, sincroniza la memoria compartida para que un thread no escriba en dicha memoria mientras otros puedan estar operando en ella. Esto, evidentemente, tiene un costo mayor en tiempo de ejecución.

Elementos comunes

Ambas clases heredan de AbstractStringBuilder y es esta clase la que almacena en un vector los caracteres que pertenecen al String que manejan las subclases.

StringBuffer esta “a salvo de threads” y es su principal diferencia respecto de StringBuilder. Por ejemplo, se puede apreciar la diferencia de declaración para un mismo método.

StringBuffer

```
public synchronized StringBuffer append(String str) {
      super.append(str);
      return this;
}
```

StringBuilder

```
public StringBuilder append(Object obj) {
      return append(String.valueOf(obj));
}
```

Si bien la sincronización existe es sumamente extraño que se tenga necesidad de ella. Cuando dos threads compiten por el acceso a memoria compartida (race condition) difícilmente lo hagan dentro del contenido de AbstractStringBuilder que utilizan ambas clases y es poco probable que surjan problemas de sincronización. En cambio, debido al mecanismo de sincronización que posee una clase respecto de la otra, si hay un efecto visible en el rendimiento. Diferentes verificaciones encontraron que cuando se realizan más de 1000 operaciones sobre ambas clases, StringBuilder puede ser hasta un 30% más rápido.

Ejemplo de uso

```
String texto="";
System.out.println(texto+"\t"+texto.hashCode());
texto+="h";
```

```
System.out.println(texto+"\t"+texto.hashCode());
texto+="o";
System.out.println(texto+"\t"+texto.hashCode());
texto+="l";
System.out.println(texto+"\t"+texto.hashCode());
texto+="a";
System.out.println(texto+"\t"+texto.hashCode());
```

La salida en consola es la siguiente:

```
      0
h     104
ho    3335
hol   103493
hola  3208380
```

Debido al concepto de inmutabilidad de la clase String, cada vez que se asigna un nuevo valor al String, se genera un nuevo objeto, cambiando su hashCode. Esto representa un tiempo de ejecución innecesario cuando se cambian muchas veces los valores de un String, por ejemplo cuando se lee un archivo carácter a carácter y se acumula en un String.

```
StringBuilder sb=new StringBuilder();
System.out.println(sb+"\t"+sb.hashCode());
sb.append("h");
System.out.println(sb+"\t"+sb.hashCode());
sb.append("o");
System.out.println(sb+"\t"+sb.hashCode());
sb.append("l");
System.out.println(sb+"\t"+sb.hashCode());
sb.append("a");
System.out.println(sb+"\t"+sb.hashCode());

System.out.println(sb.reverse());
```

La salida en consola es la siguiente:

```
      1915318863
h     1915318863
ho    1915318863
hol   1915318863
hola  1915318863
aloh
```

Al contener una cadena de texto dentro de un StringBuilder, se puede notar que se contiene en un mismo objeto su hashCode no cambia, mejorando de esta forma la performance.

La clase StringTokenizer

El procesamiento de texto a menudo consiste en analizar una cadena de entrada que posee un formato. El análisis por interpretación es la división del texto en un conjunto de partes discretas, o cortes (tokens), que en una cierta secuencia puede tener un significado semántico. La clase StringTokenizer proporciona el primer paso en este proceso de análisis, a menudo llamada el analizador de léxico (análisis lexicográfico) o escáner. StringTokenizer implementa la interfaz

Enumeration. Por lo tanto, dada una cadena de entrada, se pueden enumerar los tokens individuales contenidos en la misma utilizando StringTokenizer.
Para usar StringTokenizer, se especifica una cadena de entrada y una cadena que contiene delimitadores. Los delimitadores son los caracteres que se usan para separar los elementos. Cada caracter en la cadena de delimitadores se considera uno válido, por ejemplo, ",;:" establece los delimitadores de una coma, punto y coma, y dos puntos. El conjunto predeterminado de delimitadores se compone de los caracteres que implican espacios en blanco: espacio, tabulador, nueva línea y retorno de carro.

Los constructores StringTokenizer se muestran a continuación:

```
public StringTokenizer(String str)
public StringTokenizer(String str, String delim)
public StringTokenizer(String str, String delim, boolean returnDelims)
```

En todas las versiones, str es la cadena que se corta. En la primera versión, se utilizan los delimitadores por defecto. En las versiones segunda y tercera, los delimitadores son una cadena que los especifica.
Los delimitadores no se retornan como tokens por los dos primeros constructores. Una vez que se haya creado un objeto StringTokenizer, el método nextToken se utiliza para extraer tokens consecutivos. El método hasMoreTokens retorna verdadero mientras haya más tokens para ser extraídos. Como StringTokenizer implementa una enumeración, se implementan los métodos los hasMoreElements y nextElement, y actúan de la misma forma que lo hacen hasMoreTokens y nextToken, respectivamente.

```
//Ejemplode código
String texto="Hola a todos, esto es una prueba";

//Selecciones un constructor
//StringTokenizer st=new StringTokenizer(texto);
//StringTokenizer st=new StringTokenizer(texto,",");
StringTokenizer st=new StringTokenizer(texto,",",true);
System.out.println("Cantidad de Tokens: "+st.countTokens());

System.out.println("--------------------");
while(st.hasMoreTokens()){
      System.out.println(st.nextToken());
}

//Recorrido con elements
//System.out.println("--------------------");
//while(st.hasMoreElements()){
//    System.out.println(st.nextElement());
//}
```

Expresiones en java

Expresiones regulares

¿Qué son las expresiones regulares?

Las expresiones regulares son una forma de describir un conjunto de cadenas sobre la base de características comunes compartidas por cada cadena en dicho conjunto. Pueden ser utilizadas para buscar, editar o manipular el texto y datos. Se puede aprender una sintaxis específica para crear expresiones regulares pero esto va más allá de la sintaxis normal del lenguaje de programación Java. Las expresiones regulares varían en complejidad, pero una vez que se entienden

los conceptos básicos de cómo están construidas, se es capaz de descifrar (o crear) una expresión regular.
Se verá la sintaxis de expresiones regulares con la ayuda de la java.util.regex API. En el mundo de las expresiones regulares, hay muchos sabores diferentes para elegir, tales como grep, Perl, Tcl, Python, PHP, y awk. La sintaxis de expresiones regulares en el java.util.regex API es muy similar a la que se encuentra en Perl.
¿Cómo son representadas las expresiones regulares en el paquete java.util.regex?
El paquete java.util.regex se compone fundamentalmente de tres clases: Pattern , Matcher y PatternSyntaxException .

Pattern: un objeto de este tipo es una representación compilada de una expresión regular.
La clase Pattern no tiene constructores públicos. Para crear un patrón, primero se debe invocar a alguno de sus métodos public static compile, que luego devuelven un objeto del tipo Pattern. Estos métodos aceptan una expresión regular como primer argumento.

Matcher: un objeto de este tipo es el motor que interpreta al patrón y realiza operaciones de reconocimiento sobre una cadena de entrada. Al igual que Pattern la clase, Matcher no define constructores públicos. Se obtiene un objeto del tipo Matcher mediante la invocación del método matcher en un objeto del tipo Pattern.

PatternSyntaxException: un objeto de este tipo es una excepción no verificada que indica un error de sintaxis en un patrón de la expresión regular.

Cadena de literales
La forma más básica de coincidencia de patrones con el apoyo de esta API es la búsqueda en una cadena literal. Por ejemplo, si la expresión regular es “hola” y la cadena de entrada es “hola”, entonces la búsqueda tendrá éxito porque ambas cadenas son idénticas.

Ejemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class VerificaExpresion {
      public static void main(String[] args) {
            String expresion = "hola";
            String cadena = "hola";
            boolean encontrado = false;
            Pattern patron = Pattern.compile(expresion);
            Matcher coincidencias = patron.matcher(cadena);
            int lugar = 0;
            while (coincidencias.find(lugar)) {
                  System.out.printf("Se encontró el texto \"%s\" " +
                        "comenzando en el "
                        + "índice %d y finalizando en el índice %d.%n",
                        coincidencias.group(), coincidencias.start(),
                        coincidencias.end());
                        lugar = coincidencias.end();
                        encontrado = true;
            }
            if (!encontrado) {
                  System.out.printf("No se encontró correspondencia.");
            }
            encontrado = false;
      }
}
```

Una versión más completa del método compile, que recibe dos argumentos, permite controlar más de cerca cómo el modelo se aplica en la búsqueda de una coincidencia. El segundo argumento

es un valor de tipo int que especifica uno o más de las siguientes banderas definidas en la clase Pattern:

CASE_INSENSITIVE: Coincidencias ignorando mayúsculas o minúsculas, pero asume sólo se buscan coincidencias con caracteres del tipo US-ASCII.

MULTILINE: Permite encontrar como coincidencia el principio o final de las líneas. Sin esta bandera, sólo el comienzo y el final de la secuencia entera se consideran una coincidencia.

UNICODE_CASE: Cuando esto se especifica, junto con CASE_INSENSITIVE , las coincidencias sin tener en cuenta mayúsculas y minúsculas son consistentes con el estándar Unicode.

DOTALL: Hace que la expresión coincida con cualquier caracter, incluyendo terminaciones de línea.

LITERAL: Hace que la cadena que especifica el patrón sea tratado como una secuencia de caracteres literales, por lo que las secuencias de escape, por ejemplo, no son reconocidas como tales.

CANON_EQ: Coincidencias que tienen en cuenta la equivalencia canónica de caracteres combinados. Por ejemplo, algunos caracteres que tienen signos diacríticos pueden ser representados como un caracter único o como un solo carácter con un diacrítico seguido de un carácter diacrítico. Este indicador trata esto como una coincidencia.

COMMENTS: Permite espacios en blanco y comentarios en un patrón. Los comentarios en un patrón comienzan con # hasta el final de la línea.

UNIX_LINES: Activa el modo de líneas de UNIX, donde sólo el '\ n' es reconocido como un terminador de línea.

UNICODE_CHARACTER_CLASS: Activa la versión Unicode de clases de caracteres predefinidos.

Todas estas banderas tienen activado un único bit dentro de un valor de tipo int para que se puedan combinar con el operador OR o por simple adición. Por ejemplo, se puede especificar las banderas CASE_INSENSITIVE y UNICODE_CASE con la siguiente expresión:

Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE
O también se puede escribir como:
Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE

Se debe tener cuidado cuando se quiere añadir una bandera a una variable que representa un conjunto existente de éstas.
Si la bandera ya existe, produce un resultado erróneo debido al acarreo en la adición de los dos bits correspondientes al siguiente bit. Utilizar el operador OR siempre produce el resultado correcto.
Si se desea hacer coincidir "hola" ignorando mayúsculas y minúsculas, se puede crear el patrón con la siguiente declaración:

```
Pattern patron = Pattern.compile("hola", Pattern.CASE_INSENSITIVE);
```

Además de la excepción producida por la primera versión del método, esta versión arroja una

IllegalArgumentException si el segundo argumento tiene valores de bits establecidos que no corresponden a ninguna de las constantes de bandera definidas en la clase de Pattern.

Comparadores

Después de tener un objeto patrón, se puede crear un objeto del tipo Matcher en el cual buscar una cadena especifica, como por ejemplo:

String oracion = "Juan, dice a Pedro hola hola ‘hola’, hola hola ‘hola hola’.";

Matcher matchHola = patron.matcher(oracion);

La primera declaración define la cadena de la oración en la que se desea buscar. Para crear el objeto del tipo Matcher, se llama al método matcher en el objeto patrón con la cadena que se va a analizar como argumento.
Esto retorna un objeto Matcher que puede analizar la cadena que se le ha pasado. El parámetro para el método matcher es en realidad de tipo CharSequence. Esta es una interfaz que es implementada por las clases String, StringBuffer y StringBuilder, por lo que se puede pasar una referencia a cualquiera de estos tipos como argumento del método. La clase java.nio.CharBuffer también implementa CharSequence, por lo que también se puede pasar el contenido de un objeto de tipo CharBuffer al método. Esto significa que si se utiliza un CharBuffer para almacenar los datos de caracteres leídos desde un archivo, se los puede pasar directamente al método matcher.
Una de las ventajas de la implementación de Java de las expresiones regulares es que se puede reutilizar un objeto del tipo Pattern para crear objetos del tipo Matcher para buscar el patrón en una variedad de cadenas. Para utilizar el mismo patrón para buscar en otra cadena, sólo se cambia el argumento del método matcher en el objeto patrón con la nueva cadena como argumento. De esta manera se tiene un objeto Matcher nuevo que se puede utilizar para buscar en la nueva cadena.
También se puede cambiar la cadena que un objeto Matcher utiliza para la búsqueda llamando al método reset que éste tiene con la nueva cadena como argumento. Por ejemplo:

matchHola.reset(“Hola Mundo. Esta la nueva cadena que tiene la palabra hola.”);

Esto remplaza la cadena anterior, oracion, en el objeto Matcher, de manera que ahora es capaz de buscar en la nueva cadena. Al igual que el método matcher en la clase de Pattern, el tipo de parámetro para el método reset es del tipo CharSequence, por lo que se puede pasar una referencia de tipo String, StringBuffer, StringBuilder, o CharBuffer.

Búsqueda de una cadena

Una vez que se tiene un objeto del tipo Matcher, se puede usar para buscar en la cadena. Al llamar al método find del objeto del tipo Matcher se busca en la cadena la siguiente ocurrencia del patrón. Si se encuentra el patrón, el método almacena información acerca de dónde se encuentra en el objeto Matcher y devuelve true. Si no encuentra el patrón, devuelve false. Cuando el patrón se ha encontrado, llamando al método start del objeto Matcher se obtiene la posición del índice en la cadena donde se encontró el primer carácter del patrón. Al llamar al método end retorna la posición del índice después del último carácter del patrón. Los dos valores del índice se devuelven como tipo int .

Tener en cuenta que no se debe llamar a start o end en el objeto de tipo Matcher antes de haber tenido éxito en encontrar el patrón. Hasta que un patrón haya sido encontrado, el objeto del tipo Matcher se encuentra en un estado indefinido y llamar a cualquiera de estos métodos da como resultado una excepción IllegalStateException.

Por lo general, se quiere encontrar todas las ocurrencias de un patrón en una cadena. Cuando se llama al método find, la búsqueda se inicia ya sea desde el comienzo de la región de este comparador, o en el primer carácter no registrado por una llamada previa a find. Por lo tanto, se pueden encontrar fácilmente todas las ocurrencias del patrón mediante la búsqueda en un bucle.
Código de Ejemplo:

```
// Una expresión regular y una cadena en la cual buscar
String expresionRegular = "hola";
String oracion = "Juan, dice a Pedro hola hola ‘hola’, hola hola ‘hola
      hola’.";
// Se marcan los lugares donde se encuentra el patrón
// Funciona bien si la letra es siempre del mismo tamaño
char[] marcador = new char[oracion.length()];
// Llenar la cadena de espacios para luego colocar el caracter que marca
Arrays.fill(marcador, ' ');
// Obtener el compador requerido
Pattern patron = Pattern.compile(expresionRegular);
Matcher comparador = patron.matcher(oracion);
// Buscar cada coincidencia y marcarla
while (comparador.find()) {
      System.out.println("Patrón encontrado. Comienza en: "
      + comparador.start() + " y termina en " + comparador.end());
      Arrays.fill(marcador, comparador.start(), comparador.end(), '^');
}
// Mostrar el resultado con las marcas generadas
System.out.println(oracion);
System.out.println(marcador);
```

Definiendo un conjuntos de caracteres

Una expresión regular puede estar formada por caracteres ordinarios, la cuales son letras mayúsculas o minúsculas y números, además de secuencias de meta-caracteres, que son aquellos que tienen un significado especial. El patrón en el ejemplo anterior era sólo la palabra "hola", pero si se quiere buscar en una cadena ocurrencias de conjuntos de caracteres que se encuentren contiguos unos con otros, se debe especificar la expresión regular de otra forma.
Una opción es para especificar un carácter intermedio como un comodín mediante el uso de un punto, el cual es un ejemplo de un meta-carácter. Este coincide con cualquier carácter, excepto el del final de línea, por lo que una expresión regular "s.l", representa cualquier secuencia de tres caracteres que comienza con "s" y termina con "l".

```
// s_l
String expresionRegular = "s.l";

//s[vocal]l
String expresionRegular = "s[aeiou]l";
```

[aeiou]: Este es un ejemplo simple en el cual cualquier caracter que sea una vocal en minúscula cumple la condición.

[^aeiou]: Esto representa cualquier carácter excepto los que aparecen entre los corchetes a la derecha de ”^“. Por lo tanto se reconoce cualquier carácter que no es una vocal minúscula.

[a-e]: Esto define un rango inclusivo simple. Cualquiera de las letras "a" a "e" en este caso .

[a-cs-zA-E]: Esto define un rango inclusivo múltiple. Esto se corresponde con cualquier carácter de "a" a "c", de "s" a "z" o de "A" a "E".

Operadores lógicos en las expresiones regulares

Usted puede utilizar el operador && para combinar las clases que definen los conjuntos de caracteres. Esto es particularmente útil cuando se utiliza en combinación con el operador de negación, ^, que aparece en la segunda fila de la tabla en la sección anterior. Por ejemplo, si se desea especificar que cualquier consonante minúscula es una coincidencia, se podría escribir la expresión regular como:

[b-df-hj-np-tv-z]

Sin embargo, esto no sólo no es claro sino que además es trabajoso. Utilizando operadores lógicos para definir la expresión es mucho más simple para definir coincidencias:

[a-z && [^ aeiou]]

Cuando se escribe una expresión como la anterior, se específica que la coincidencia válida se considera para cualquier carácter desde a hasta z pero que ADEMÁS NO sean la vocales entre los corchetes. El operador && determina la intersección de ambas condiciones al igual que lo hace en una expresión lógica del lenguaje.

Análogamente existe el operador | (OR) que funciona como el “o” incluyente. Los operadores relaciones se pueden combinar de la manera que resulte más conveniente para generar condiciones sobre los caracteres de las expresiones regulares.

Conjuntos predefinidos de caracteres

También se puede utilizar una serie de caracteres predefinidos que proporcionan una notación abreviada para los conjuntos comúnmente utilizados.

.	Representa cualquier carácter, como ya se ha visto.
\d	Representa cualquier dígito y por lo tanto, es la abreviatura de [0-9].
\D	Representa cualquier carácter que no es un dígito. Por tanto, es equivalente a [^ 0-9].
\s	Representa cualquier espacio en blanco. Se entiende como tal a una tabulación '\ t', un carácter de nueva línea '\ n', un carácter de avance de línea '\ f', un retorno de carro '\ r', o un salto de página 'x0B'.
\S	Representa cualquier carácter que no sea un blanco y por lo tanto equivalente a [^ \ s].
\w	Esto representa a un posible caracter en una palabra, que corresponde a una letra mayúscula o minúscula, un dígito o guion bajo. Por tanto, es equivalente a [a-zA-Z_0-9].
\W	Esto representa cualquier carácter que no es un caracter en una palabra, por lo que es equivalente a [^ \ w].

El tratamiento de cadenas que tiene Java obliga a tratar los caracteres especiales de las expresiones regulares de manera diferente a la que se puede realizar en otros entornos. Por ejemplo, el uso de la barra invertida en el lenguaje indica un carácter de escape, con lo cual, para indicar una barra invertida en la expresión regular, la misma se debe duplicar para que el lenguaje la interprete como

una sola cuando analiza la cadena en tiempo de compilación. De esta manera, si se quieren indicar tres dígitos, la cadena de la expresión regular tendría el siguiente formato:

“\\d\\d\\d”

Otro ejemplo importante es jugar con los espacios en blanco para determinar longitudes exactas en las palabras. En los ejemplos anteriores se buscaron coincidencias con respecto de la cadena “sol” en diferentes combinaciones, pero si se quiere buscar palabras de tres letras que empiecen con una S y terminen con L, se pueden utilizar los caracteres en blanco para delimitarla y así obtener coincidencias exactas:

“\\sS.L[\\s,]”

Límites en las coincidencias

^ Especifica el comienzo de una línea. Por ejemplo, para encontrar la palabra Java en el principio de cualquier línea se puede utilizar la expresión "^Java".

$ Especifica el final de una línea. Por ejemplo, para encontrar la palabra Java al final de cada línea se puede utilizar la expresión "Java$". Por supuesto, si se espera un punto al final de una línea de la expresión sería "Java\\.$".

\b Especifica el límite en una palabra. Para encontrar palabras de tres letras que comienzan con "s" y terminan con 'l', se puede utilizar la expresión "\\bs.l\\b".

\B Las palabras no son un límite - el complemento de \b.

\A Especifica el inicio de la cadena buscada. Para encontrar la palabra El, al principio de una cadena en la que se busca, se puede utilizar la expresión "\\AEl\\b". El \\b en el extremo de la expresión regular es necesario para evitar encontrar una coincidencia como Elegido o Elías al comienzo de la cadena.

\z Especifica el final de una cadena que se busca. Para encontrar la palabra “eliminación” seguida de un punto al final de una cadena, se puede usar la expresión "\\beliminación\\.\\z".

\Z El extremo final de la cadena a analizar, excepto para el carácter terminador final. Un terminador final es un carácter de nueva línea ('\n') si se establece Pattern.UNlx_LlNES. De lo contrario, también puede ser un retorno de carro ('\r'), un retorno de carro seguido de un carácter de nueva línea, un carácter de nueva línea ('\u0085'), un separador de línea ('\u2028'), o un separador de párrafo ('\u2029').

Uso de cuantificadores

Los cuantificadores permiten identificar caracteres que se repiten según un formato especificado creando una secuencia de caracteres. Por ejemplo, un valor numérico es una secuencia de números que se puede describir mediante un cuantificador. Si se quisiera buscar un valor numérico en una cadena, por ejemplo un entero que es el caso más simple, se puede utilizar como cuantificador el meta caracter “+”. De esta manera, sumando a que cada dígito se puede especificar la expresión regular como”\\d”, una secuencia de dígitos se puede especificar como “\\d+”.

Para casos más complejos, un número puede incluir una coma decimal. Sin embargo, los números decimales pueden incluir o no una coma decimal, pero será una única ocurrencia. Por lo tanto esta puede ocurrir una vez o ninguna por cada secuencia de números que se analice. Además, la secuencia poseerá números luego de la coma. Esto se puede resolver mediante el meta caracter "?" y generar una expresión regular como la siguiente:

"\\d+.?\\d+"

Se puede leer la expresión anterior como: "se busca una secuencia de números en la cual puede o no aparecer un punto y luego de él se pueden encontrar más dígitos". Esta expresión se puede mejorar aún más. Como la parte decimal puede ser opcional, se pueden utilizar paréntesis como meta caracteres y combinarlos con el signo de pregunta para indicar que su aparición puede ser opcional:

"\\d(\\.\\d+)?"

Esta expresión indica que una coincidencia válida es si la cadena posee una secuencia de caracteres numéricos y también si opcionalmente aparece un punto seguido de más dígitos. Para mejorar la capacidad de encontrar coincidencias, se puede agregar que se consideren válidas las ocurrencias con signo, aparezca el mismo o no. De esta manera, la expresión regular queda:

"[+|-]?\\d+(\\.\\d+)?"

Si la cadena tiene un número que empieza con una coma (esto es, da la parte entera como un cero supuesto por la omisión, ej: ,23 en lugar de 0,23), la expresión anterior no sirve. En este caso hay que considerar que:

Puede empezar con un signo
Puede empezar con una coma
Puede ser un número sin parte decimal
Puede ser un número con parte decimal y coma
Para que un número empiece con coma y signo, la expresión regular seria:

"[+|-]?\\.\\d+"

Esta expresión encontraría como ocurrencia aquellos números que comiencen con una coma y muestres sólo su parte decimal. Se debe combinar esta opción con la anterior para cumplir con todos los puntos enumerados:

"[+|-]?(\\d+(\\.\\d+)?)|\\.\\d+)"

Notar que se resolvió como si fuera una expresión aritmética para combinar ambas expresiones. Se saco el factor común que es la resolución del signo fuera de los paréntesis y luego se colocó ambas expresiones unidas mediante el operador lógico OR.

```
Codigo de Ejemplo

String regEx = "[+|-]?(\\d+(\\.\\d+)?)|(\\.\\d+)";
String str= "256 es el cuadrado de 16 y -2.5 al cuadrado es 6.25 y -.243 es
            menor que 0.1234.";
Pattern pattern = Pattern.compile(regEx);
Matcher m = pattern.matcher(str);
while (m.find ()) {
      System.out.println(m.group());
```

```
}

Salida
256
16
-2.5
6.25
.243
0.1234
```

Expresiones Lambdas

Introducción

A partir de JDK 8, se agregó una característica a Java que mejoró profundamente el poder expresivo del lenguaje. Esta característica es la **expresión lambda** (Lambda Expressions). Las expresiones lambda no solo agregaron nuevos elementos de sintaxis al lenguaje, sino que también simplificaron la forma en que se implementan ciertas construcciones comunes. De la misma manera que la adición de genéricos reformó Java hace años, las expresiones lambda continúan remodelando Java hoy en día. Ellos realmente son tan importantes.

La adición de expresiones lambda también proporcionó el catalizador para otras características de Java. Ya ha visto uno de ellos, el **método default** (Leer Métodos de Interfaces en Java), que le permite definir el comportamiento predeterminado para un método de interfaz.

Más allá de los beneficios que las expresiones lambda aportan al lenguaje, existe otra razón por la cual constituyen una parte tan importante de Java. En los últimos años, **las expresiones lambda se han convertido en un foco principal del diseño de lenguaje de computadora**. Por ejemplo, se han agregado a lenguajes como **C#** y **C++**. Su inclusión en Java lo ayuda a seguir siendo el lenguaje vibrante e innovador que los programadores esperan.

Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase.
su sintaxis básica se detalla a continuación:

```
( parámetros ) -> { cuerpo-lambda }
```

1. El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.
2. 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.
3. Cuerpo de lambda:
   - •Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan 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 .

Algunos ejemplos de expresiones **lambda** pueden ser:

```
• z -> z + 2
• () -> System.out.println(” Mensaje 1 “)
• (int longitud, int altura) -> { return altura * longitud; }
• (String x) -> {
        String retorno = x;
        retorno = retorno.concat(” ***”);
        return retorno;
 }
```

Como hemos visto las expresiones lambda son funciones anónimas y pueden ser utilizadas allá donde el tipo aceptado sea una interfaz funcional.

La clave para entender la expresión lambda son dos conceptos. El primero es la **expresión lambda**, en sí misma. El segundo es la **interfaz funcional.** Comencemos con una definición simple de cada uno.

- **Una expresión lambda es, esencialmente, un método anónimo** (es decir, sin nombre). Sin embargo, este método no se ejecuta solo. En cambio, se usa para implementar un método definido por una interfaz funcional. Por lo tanto, una expresión lambda da como resultado una forma de clase anónima. Las expresiones lambda también se conocen comúnmente como **cierres** (closures).
- **Una interfaz funcional es una interfaz que contiene uno y solo un método abstracto**. Normalmente, este método especifica el propósito previsto de la interfaz. Por lo tanto, una interfaz funcional típicamente representa una sola acción. Por ejemplo, la interfaz estándar **Runnable** es una interfaz funcional porque define solo un método: **run()**. Entonces, *run()* define la acción de *Runnable*. Además, una interfaz funcional define el tipo de objetivo de una expresión lambda. Aquí hay un punto clave: una expresión lambda solo se puede usar en un contexto en el que se especifica un tipo de objetivo. Otra cosa: una interfaz funcional a veces se conoce como tipo SAM, donde SAM significa **Single Abstract Method**.

Una interfaz funcional puede especificar cualquier método público definido por **Object**, como **equals()**, sin afectar su estado de “interfaz funcional”. Los métodos de objetos públicos se consideran miembros implícitos de una interfaz funcional porque se implementan automáticamente mediante una instancia de una interfaz funcional.

Ejemplo de Interfaz funcional.

Aquí hay un ejemplo de una interfaz funcional:

```
interface MiValor{
      double getValor();
}
```

En este caso, el método *getValor()* es implícitamente abstracto, y es el único método definido por *MiValor*. Por lo tanto, *MiValor* es una interfaz funcional, y su función está definida por *getValor()*.

Como se mencionó anteriormente, una expresión lambda no se ejecuta por sí misma. Más bien, forma la implementación del método abstracto definido por la interfaz funcional que especifica su

tipo de objetivo. Como resultado, una expresión lambda solo se puede especificar en un contexto en el que se define un tipo de objetivo. Uno de estos contextos se crea cuando una expresión lambda se asigna a una referencia de interfaz funcional. Otros contextos de tipo de objetivo incluyen **inicialización de variable**, **declaraciones de return** y **argumentos de método**, por nombrar algunos.

Vamos a trabajar a través de un simple ejemplo. Primero, una referencia a la interfaz funcional:

MiValor está declarado:

```
// Crea una referencia a una instancia de MiValor.
MiValor miVal;
```

A continuación, se asigna una expresión lambda a esa referencia de interfaz:

```
// Usa una lambda en un contexto de asignación.
miVal = () -> 28.6;
```

Esta expresión lambda es compatible con *getValor()* porque, al igual que *getValor()*, no tiene parámetros y devuelve un resultado **double**. En general, el tipo de método abstracto definido por la interfaz funcional y el tipo de expresión lambda debe ser compatible. Si no lo son, se producirá un error en tiempo de compilación.

Como probablemente pueda adivinar, los dos pasos que se muestran pueden combinarse en una sola declaración, si lo desea:

```
MiValor miVal = () -> 28.6;
```

Aquí, *miVal* se inicializa con la expresión lambda.

Cuando se produce una expresión lambda en un contexto de tipo de objetivo, se crea automáticamente una instancia de una clase que implementa la interfaz funcional, definiendo la expresión lambda el comportamiento del método abstracto declarado por la interfaz funcional. Cuando se llama a ese método a través del objetivo, se ejecuta la expresión lambda. Por lo tanto, **una expresión lambda nos da una forma de transformar un segmento de código en un objeto**.

En el ejemplo anterior, la expresión lambda se convierte en la implementación del método *getValor()*. Como resultado, lo siguiente muestra el valor 2*8.6*:

```
// Llama a getValor(), que se implementa mediante la expresión lambda
// asignada previamente.
System.out.print1n("Un valor constante: " + miVal.getValor());
```

Debido a que la expresión lambda asignada a *miVal* devuelve el valor 28.6, ese es el valor obtenido cuando se llama a *getValor()*.

Expresión lambda con uno o más parámetros.

Si la expresión lambda toma uno o más parámetros, entonces el método abstracto en la interfaz funcional también debe tomar el mismo número de parámetros. Por ejemplo, aquí hay una interfaz funcional llamada *MiValParam*, que le permite pasar un valor a *getValor()*:

```
interface MiValParam {
      double getValor(double v);
```

```
}
```

Puede usar esta interfaz para implementar la lambda recíproca. Por ejemplo:

```
MiValParam miValor = (n) -> 1.0 / n;
```

Entonces puede usar *miValor* como:

```
System.out.println("El recíproco de 4.0 es: "+miValor.getValor(4.0));
```

Aquí, *getValor()* se implementa mediante la expresión lambda a la que se refiere *miValor*, que devuelve el recíproco del argumento. En este caso, 4.0 se pasa a *getValor()*, que devuelve 0.25.

Hay algo más de interés en el ejemplo anterior. Tenga en cuenta que el tipo de *n* no está especificado. Más bien, su tipo se deduce del contexto. En este caso, su tipo se deduce del tipo de parámetro de *getValor()* tal como lo define la interfaz *MiValParam*, es **double**. También es posible especificar explícitamente el tipo de un parámetro en una expresión lambda. Por ejemplo, esta es también una forma válida de escribir lo anterior:

```
(double n) -> 1.0 / n;
```

Aquí, **n** se especifica explícitamente como **double**. Por lo general, no es necesario especificar explícitamente el tipo.

Antes de continuar, es importante destacar un punto clave: para que una expresión lambda se use en un contexto de tipo de objetivo, **el tipo de método abstracto y el tipo de expresión lambda deben ser compatibles.** Por ejemplo, si el método abstracto especifica dos parámetros *int*, entonces la lambda debe especificar dos parámetros cuyo tipo sea explícitamente *int* o inferirse implícitamente como *int* por el contexto. En general, el tipo y el número de los parámetros de la expresión lambda deben ser compatibles con los parámetros del método y su tipo de retorno.

Bloque Expresiones Lambda

El cuerpo de las lambdas que se muestra en los ejemplos anteriores consiste en una sola expresión. Estos tipos de cuerpos lambda se denominan **cuerpos de expresión**, y las lambdas que tienen cuerpos de expresión a veces se denominan **expresión lambdas**. En un cuerpo de expresión, el código en el lado derecho del operador lambda debe consistir en una sola expresión, que se convierte en el valor de lambda. Aunque la expresión lambdas es bastante útil, a veces la situación requerirá **más de una sola expresión.**

Para manejar estos casos, Java admite un segundo tipo de expresión lambda en la cual el código en el lado derecho del operador lambda consiste en un **bloque de código** que puede contener más de una declaración. Este tipo de cuerpo lambda se llama **cuerpo de bloque**. Las lambdas que tienen cuerpos de bloque a veces se denominan **bloque lambdas**.

Un bloque lambda amplía los tipos de operaciones que se pueden manejar dentro de una expresión lambda porque permite que el cuerpo de la lambda contenga **múltiples declaraciones.** Por ejemplo, en un bloque lambda puede declarar variables, usar bucles, especificar sentencias **if** y **switch**, crear bloques anidados, y así sucesivamente. Un bloque lambda es fácil de crear. Simplemente encierre el cuerpo entre llaves como lo haría con cualquier otro bloque de declaraciones

Además de permitir declaraciones múltiples, las lambdas de bloque se usan de forma muy parecida a la expresión lambdas que acabamos de mencionar. Sin embargo, una diferencia clave es que debe usar explícitamente una declaración **return** para devolver un valor. Esto es necesario porque un bloque de cuerpo lambda no representa una sola expresión.

Ejemplo de Expresión Lambda en bloque

```
// Un bloque lambda que encuentra el divisor positivo
// más pequeño de un valor int.
interface FuncNum {
      int func (int n);
}
class LambdaDemo{
      public static void main(String[] args) {
            // Este bloque lambda devuelve el divisor positivo más pequeño
            // de un valor
            FuncNum divPeq= (n) ->{
                  int res=1;
                  // Obtenga el valor absoluto de n.
                  n = n<0 ? -n:n;
                  for (int i=2; i<=n/i;i++)
                        if ((n%i)==0) {
                              res = i;
                              break;
                        }
                        return res;
            };
            System.out.println("El divisor más pequeño de 12 es:
                  "+divPeq.func(12));
            System.out.println("El divisor más pequeño de 15 es:
                  "+divPeq.func(-15));
      }
}
```

Ejemplos de uso

RECORRER UNA LISTA DE NÚMEROS EN VERSIONES ANTERIORES DE JAVA

```
for (Integer numero : Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)) {
      System.out.print(numero + " ");
}
```

RECORRER UNA LISTA DE NÚMEROS UTILIZANDO EXPRESIONES LAMBDA EN JAVA

```
// imprimir una lista utilizando expresiones lambda en Java 8
Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).forEach(n -> System.out.print(n + " "));

// Parámetro Item implícito en Lamda
Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).forEach(System.out::print);
// En este caso el parámetro n es implícito no se declara, pero el método
// que usa el nuevo operador :: (cuatro puntos) usa ese parámetro implícito
```

Como se puede ver el cambio entre versiones anteriores y Java 8 es bastante bueno de hecho se puede hacer en 2 líneas de código todo lo que en versiones anteriores se lo hace entre 9 y 12 líneas.

API Reflect

Introducción al “lado oscuro de java”.

Una de las funcionalidades más potentes y poco conocidas de Java es su soporte para reflexión. Mediante la Java Reflection API el programador puede inspeccionar y manipular clases e interfaces (así como sus métodos y campos) en tiempo de ejecución, sin conocer a priori (en tiempo de compilación) los tipos y/o nombres de las clases específicas con las que está trabajando.

Quizás pueda parecernos en una primera impresión una funcionalidad con usos limitados. Pero debemos saber que, por ejemplo, muchos frameworks de alto nivel como Hibernate, Spring o Tapestry hacen un uso extensivo de esta API para facilitarle la vida al programador al permitirle que use simples clases POJO para trabajar con ellas. Otros frameworks menos potentes (o versiones antiguas de estos mismos frameworks), obligaban al programador a que sus clases implementaran ciertos interfaces o pertenecieran a complicadas jerarquías de clases, lo cual limitaba la flexibilidad del programador y complicaba la comprensión del código.

Clase

En Java todas las clases son objetos de la clase Class.
Cuando en nuestro programa queremos usar reflexión para poder trabajar con un objeto del que desconocemos su tipo (en el caso de Java, esto es lo mismo que decir que desconocemos el nombre de su clase), lo primero que debemos hacer es averiguar la clase a la que pertenece. En situaciones normales conoceremos el tipo de un objeto en tiempo de compilación, con lo que obtendríamos su clase de la forma:

```
Class userClass = User.class;
```

En aquellas situaciones en las que no conozcamos el nombre de una clase en tiempo de compilación, podemos obtener su clase en tiempo de ejecución a partir de un String que contenga el nombre completo de la clase (incluyendo los paquetes, lo que se conoce como nombre de clase totalmente calificado) usando la función Class.forName():

```
Class userClass = Class.forName("entities.Persona");
```

En nuestro ejemplo, la clase es Persona que pertenece al paquete entities.

Si por algún motivo la clase no existiese, se lanzaría una excepción ClassNotFoundException para indicarlo.

En sentido inverso, teniendo la clase de un objeto también podremos obtener una cadena con el nombre de la clase:

```
String className = userClass.getName();
```

Este ejemplo nos devuelve el nombre totalmente calificado de la clase. Si queremos sólo el nombre simple (sin el paquete), debemos usar el método getSimpleName().

Información adicional
Podemos obtener el paquete de una clase mediante:

```
Package userPackage = userClass.getPackage();
```

Nótese que no devuelve un simple String con el nombre del paquete, sino un objeto de tipo Package que contiene información sobre el paquete y métodos para su manipulación.

También podemos acceder a la superclase de una clase (que será nuevamente un objeto Class, de forma que podemos seguir haciendo reflexión sobre él) mediante el método:

```
Class userSuperclass = userClass.getSuperclass();
```

Podemos obtener una lista de los interfaces que implementa una clase (en forma de array de objetos Class) mediante:

```
Class[] userInterfaces = userClass.getInterfaces();
```

Debe tenerse en cuenta que sólo se incluyen los interfaces implementados directamente por esta clase, no los que se heredan porque son implementados por alguna superclase de la jerarquía.

Constructores

Podemos acceder a la lista de todos los constructores de una clase mediante el método:

```
Constructor[] userConstructors = userClass.getConstructors();
```

Si conocemos los tipos de los parámetros de un constructor en particular, podemos acceder a ese constructor concreto sin tener que recorrer toda la lista. Esto se hace mediante el método getConstructor(), que admite como parámetros un array con los tipos específicos de ese constructor (en el mismo orden en el que están declarados). Por ejemplo, si sabemos que nuestra clase de ejemplo contiene un constructor que acepta los parámetros (String, String, Integer) podemos acceder a él mediante:

```
Constructor userConstructor = userClass.getConstructor(new Class[]
      {String.class, String.class, Integer.class});
```

Si no existiera un constructor con esos parámetros, se lanzaría una excepción de tipo NoSuchMethodException.

A la inversa, para un constructor también podemos obtener sus parámetros mediante:

```
Class[] params = userConstructor.getParameterTypes();
```

Una vez hallamos conseguido el constructor deseado, podemos instanciar un objeto mediante el método newInstance() con los parámetros adecuados. En nuestro ejemplo:

```
Constructor userConstructor = userClass.getConstructor(new Class[]
      {String.class, String.class, Integer.class});
User user = (User) userConstructor.newInstance("Antonio González",
      "agonzalez@mimail.com", 12);
```

Obtener y manipular los atributos y métodos de una clase.

Como ejemplo que usaremos durante nuestra explicación, definimos la siguiente clase simple:

```
package com.test.model;
public class User() {
      private String alias = null;
      public String name;
      public String address;

      public User(String name) {
            this.name = name;
      }

      public setAlias(String alias) {
            this.alias = alias;
```

```
	}

	public getAlias() {
		if (alias == null) {
			return name;
		} else		{
			return alias;
		}
	}
}
```

Y supongamos que tenemos la clase almacenada en la variable userClass:

```
	Class userClass = Class.forName("com.test.model.User");
```

Atributos

Para acceder a los atributos públicos de una clase tenemos dos posibilidades. Si conocemos el nombre del atributo usaremos la siguiente instrucción:

```
Field userField = userClass.getField("name");
```

Como en los casos anteriores se nos devuelve toda la información sobre el atributo mediante una instancia al objeto correspondiente, en este caso de tipo Field.

Si la clase no tuviera ningún atributo público con ese nombre, el método getField() lanzará una excepción NoSuchFieldException.

Si por cualquier motivo no conocemos los nombres de los atributos, tenemos una forma de obtener todos los atributos públicos de una clase de la siguiente forma:

```
Field[] userFields = userClass.getFields();
```

La instrucción getFields() nos devuelve un array con un elemento de tipo Field por cada uno de los atributos públicos de la clase (en nuestro ejemplo, devolvería un array de 2 elementos: "name" y "address").

Las dos instrucciones mencionadas hasta el momento sirven sólo para acceder a los atributos públicos. Si lo que queremos es acceder a cualquier atributo (incluidos los privados), necesitaremos usar los métodos Class.getDeclaredField(String name) (para acceder sabiendo el nombre del atributo) y Class.getDeclaredFields() (que nos devolverá un array con todos los atributos declarados en la clase: "alias", "name" y "address").

Debemos tener en cuenta que estos métodos sólo nos permiten acceder a los atributos declarados expresamente en la clase en cuestión, nunca aquellos declarados en superclases de la misma.

Ahora que disponemos de una instancia Field para un atributo, podemos proceder a manipularlo. Primero veremos cómo obtener información sobre el mismo. Podemos obtener el nombre del atributo mediante:

```
String fieldName = userField.getName();
```
También podemos averiguar el tipo del atributo mediante:

```
Object fieldType = userField.getType();
```
Si lo que nos interesa es el valor contenido en el atributo, deberemos usar:

```
Object fieldValue = userField.get(userInstance);
```
Como vemos, el método Field.get() recibe un parámetro que es la instancia del objeto en cuestión del que queremos averiguar el valor de su atributo. Si estuviéramos intentando acceder a un método estático, debemos llamar al método con null como parámetro.

Si lo que queremos es cambiar el valor del atributo, escribiremos:

```
userField.set(userInstance, value);
```
Donde value es el valor que queremos asignarle (que, evidentemente, debe ser del tipo correspondiente a ese atributo) y userInstance es la instancia del objeto al que queremos asignarle el valor. Al igual que con get(), en caso de tratarse de un atributo estático, el valor de userInstance debe ser null.

Como parece lógico, los métodos anteriores (get() y set()) sólo nos permiten manipular los atributos públicos de un objeto (aquellos a los que tengamos acceso desde el contexto de nuestro código). Pero Java Reflection nos permite manipular incluso aquellos atributos privados a los que no tendríamos acceso de forma normal. Para ello sólo hay que usar el método Field.setAccessible(true), que deshabilita los chequeos de acceso para ese campo en particular (para Java Reflections sólo). De esta forma, si en nuestro ejemplo queremos modificar el valor del atributo privado "alias", sólo tendremos que hacer:

```
Object userInstance = userClass.getConstructor(new Class[]
{String.class}).newInstance(new Object[] {"José González"});
Field aliasField = userClass.getDeclaredField("alias");
aliasField.setAccessible(true);
aliasField.set(userInstance, "Pepe");
```

Métodos

De forma totalmente análoga a como accedimos a los atributos de una clase, podemos acceder a todos los métodos públicos de una clase mediante:

```
Method[] userMethods = userClass.getMethods();
```
Como podemos adivinar del ejemplo, los métodos de una clase se almacenan en un objeto de tipo Method.

Para acceder a un método específico no necesitamos saber sólo su nombre, si no que necesitamos saber el tipo y orden de los parámetros necesarios para su invocación (puesto que recordemos que la sobrecarga de operadores nos permite definir varios métodos con el mismo nombre y distintos parámetros). Así, para obtener el método setAlias() de nuestra clase de ejemplo, escribiríamos:

```
Method userMethod = userClass.getMethod("setAlias", new Class[] {String.class});
```
De la misma forma que ocurría con los atributos, estas instrucciones sólo nos devuelven los métodos públicos de una clase. Para poder acceder a los métodos privados deberemos usar respectivamente Class.getDeclaredMethod(String name) y Class.getDeclaredMethods(). También disponemos de un método Method.setAccessible(true) para poder manipular métodos privados de una clase.

Una vez tenemos el método deseado en nuestro objeto de tipo Method, procedemos a manipularlo. Para obtener el nombre y tipo de retorno usaremos métodos análogos a los que usamos para el caso de los atributos:

```
String methodName = userMethod.getName();
```

```
Object methodType = userMethod.getReturnType();
```
Finalmente, podemos invocar el método deseado mediante la orden Method.invoke():

```
Object userInstance = userClass.getConstructor(new Class[]
{String.class}).newInstance(new Object[] {"José González"});
Method setAliasMethod = userClass.getMethod("setAlias", String.class);
Method getAliasMethod = userClass.getMethod("getAlias", null);
setAliasMethod.invoke(userInstance, "Pepe");
String newAlias = getAliasMethod.invoke(userInstance, null);
```

El método Method.invoke() debe recibir como primer parámetro la instancia en particular de la que queremos invocar el método (null si es un método estático) y los parámetros del método que queremos invocar (null o array vacío si no dispone de parámetros).