**Apéndice A. Novedades en Java 9**

Novedades en Java 9

En este documento vamos a describir algunas de las principales novedades que ha aparecido en la versión Java 9, en lo que a lenguaje y APIs de uso general se refiere.

Métodos estáticos de factoría en List, Set y Map

Se trata de una serie de métodos estáticos *of()* incluidos en estas interfaces, con los que podemos crear una colección inmutable de elementos a partir de un conjunto discreto de datos.

Por ejemplo, podemos crear un List con los días de la semana de esta manera:

List<String> nombres=List.*of*("lunes","martes","miercoles");

Y mostrarlos después:

1. for(String n:nombres) {
2. System.out.println(n);
3. }

Al ser una colección inmutable, lo que no podemos es, añadir nuevos elementos o eliminar los ya existentes:

nombres.add("jueves"); //excepción UnsupportedOperationException

En el caso de Map, tenemos el método estático:

ofEntries(Entry ... entries)

Que a partir de un número variable de objetos Entry crea un mapa inmutable de los mismos. Precisamente, el método estático *entry()* de la interfaz Map, también incorporado como novedad en Java 9, permite crear objetos Entry.

El siguiente código crear una mapa de tres nombres con una clave numérica entera asociada:

1. Map<Integer,String> datos=Map.ofEntries(Map.entry(123, "Luis"),
3. Map.entry(300, "Ana"),
5. Map.entry(500, "Maria"));

Para mostrar el contenido completo de la colección podemos utilizar el método forEach(), incluido en Map desde la versión Java 8:

datos.forEach((k,v)->System.out.println("clave: "+" valor:"+v));

La interfaz Set, como List, dispone de un método estático *of(E..e)* que admite un número variable de argumentos para formar un conjunto inmutable.

Set<Integer> nums=Set.of(30,6,4,100);

Métodos privados en interfaces

Desde Java 8 las interfaces pueden incluir métodos *default* y métodos *static*, pero en ambos casos solo podían tratarse de miembros públicos. Desde Java 9  también se pueden incluir métodos privados. Estos métodos son para uso exclusivo de la interfaz, es decir, solo pueden ser llamados desde métodos *default* definidos en la propia interfaz:

1. interface Inter1{
3. //métodos privados
5. private int mayor(int a, int b){
7. return a>b?a:b;
9. }
11. private int menor(int a, int b){
13. return a<b?a:b;
15. }

18. //los métodos privados son llamados desde
19. //el método default
20. default int suma(int a, int b){
21. int s=0;
22. for(int i=menor(a,b);i<mayor(a,b);i++){
23. s+=i;
24. }
25. return s;
26. }
27. }

30. public class Test {
32. public static void main(String[] args) {
33. //implementación de la interfaz a través de
34. //una clases anónima que no incorpora nada
35. Inter1 cp=new Inter1(){};
36. System.out.println("suma "+cp.suma(10, 5));
37. }
38. }

Los métodos privados también pueden ser estáticos a fin de que puedan ser utilizados desde otros métodos estáticos de la interfaz.

Utilización de constantes efectivas en try con recursos

Antes de Java 9, si queríamos utilizar try con recursos, el objeto que representa el recurso a cerrar debía crearse dentro de los paréntesis del try:

1. try(PrintStream ps=new PrintStream(..)){
3. :
5. }

Con Java 9 es posible utilizar objetos cerrables dentro de los paréntesis del try, que hayan sido creados antes de este:

1. PrintStream ps=new PrintStream(..);
2. try(ps){
3. :
4. }

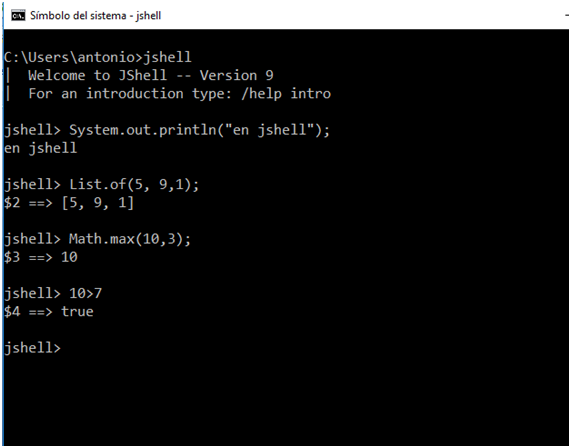
La variable ps debe ser una constante efectiva, es decir, no puede ser modificada, si eso ocurre  entonces no se podrá utilizar en el try con recursos, se producirá un error de compilación:

1. PrintStream ps=new PrintStream(..);
3. ps=new PrintStream(..);
5. try(ps){ //error de compilación
7. :
9. }

Herramienta Jshell

Java 9 incorpora la herramienta jshell, que es una aplicación basada en línea de comandos, a través de la que podemos probar de forma muy cómoda bloques de código Java, como métodos o expresiones, evitando tener que crear una clase con su método main para realizar estas tareas.

Para entrar en jshell simplemente tecleamos jshell en la línea de comandos. A partir de ahí, nos aparece una consola en la que podemos teclear cualquier código Java que devuelva o produzca un resultado :



Nuevos métodos en streams

La interfaz Stream, una de las grandes novedades que de Java 8, incorpora nuevos métodos en Java 9. Veamos algunos de los más interesantes:

* Stream<T> takeWhile(Predicate<T> p) . Para un Stream ordenado, retorna un nuevo Stream en el que se van incluyendo los elementos que cumplen con el predicado, hasta encontrar uno que no lo cumpla.

Por ejemplo, la siguiente expresión retornaría un Stream formado por los elementos 45 y 129:

1. Stream<Integer> st=Stream.of(300,45,129,260);
2. st.sorted().takeWhile(n->n<200);

En cambio, en este caso se retornaría un Stream vacío, puesto que ya el primer elemento no cumple con el predicado:

1. Stream<Integer> st=Stream.of(300,45,129,260);
2. st.sorted().takeWhile(n->n>200);

Si el Stream no está ordenado, el resultado es impredecible.

* Stream<T> dropWhile(Predicate<T> p). Funciona de forma contraría a takeWhile(), va eliminando aquellos elementos que cumplen con el predicado, hasta encontrar uno que no lo cumple.

En el siguiente ejemplo obtendríamos un Stream con los números 260 y 300:

1. Stream<Integer> st=Stream.of(300,45,129,260);
2. st.sorted().dropWhile(n->n<200)

En este otro ejemplo, el nuevo Stream contendría todos los números pues, al no cumplir el primero con el predicado, no descarta ningún elemento:

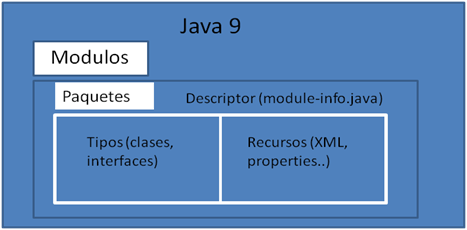
1. Stream<Integer> st=Stream.of(300,45,129,260);
2. st.sorted().dropWhile(n->n>200)

Modularidad

Una de las principales características novedosas que añade Java 9 es la modularidad, que consiste en organizar los paquetes de clases en módulos.

Podríamos decir que un módulo es un nivel de división superior al de paquete, de modo que si una clase depende de otras, no necesitemos incorporar una dependencia a un jar con cientos de clases que no vamos a necesitar, bastará con referenciar el módulo que contiene el paquete o paquetes de clases que necesitamos.

Las propias librerías del JDK están organizadas en módulos, de forma que las aplicaciones solo tengan que utilizar los módulos de las clases que necesitan, en lugar de requerir ese gran archivo jar en el que se incluían todas las miles de clases del Java SE en versiones anteriores a Java 9. Esto facilitará la creación de aplicaciones Java para dispositivos de pequeño tamaño.



Definición de un módulo

Un módulo está formado por una serie de paquetes de clases, además de un descriptor de módulo en el que se indican, por un lado, los módulos de los que depende el módulo que estamos definiendo y, por otro, los paquetes que queremos exponer a otro módulos.

Para crear un módulo, crearemos un proyecto Java estándar con sus paquetes y clases que lo van a formar. Por ejemplo, podríamos un proyecto llamado *Modularidad* con una clase *Operaciones*, dentro de un paquete com.prueba, en la que incluiremos un método *factorial()* que nos permitirá calcular el factorial de un número:

1. package com.prueba;
2. public class Operaciones {
3. public int factorial(int n) {
4. int r=1;
5. for(int i=1;i<=n;i++) {
6. r\*=i;
7. }
8. return r;
9. }
10. }

 En el *default package*, incluiremos el descriptor de paquete modulo-info.java. Para incluir este archivo, si estamos en eclipse elegiremos la opción Configure->Create module-info.java en el menú que aparece al pulsar el botón derecho del ratón sobre el proyecto. Este archivo contiene la definición del módulo, que lo llamaremos por ejemplo com.testModule:

1. module com.testModule{
3. }

Dentro del módulo debemos indicar la lista de paquetes de clases que queremos que sean accesibles desde fuera de este módulo. Esto se hace utilizando la palabra ***exports***, seguida del nombre del paquete que queremos exportar. En nuestro caso, la definición del módulo quedaría:

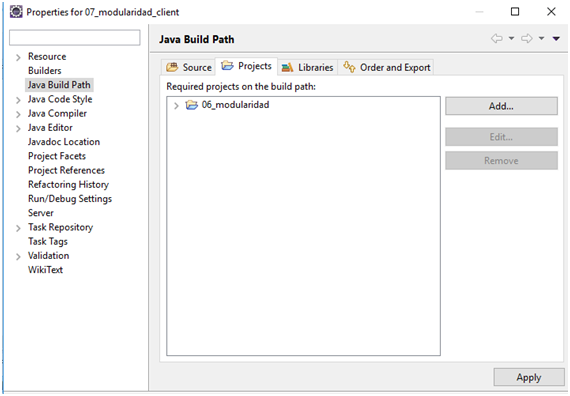
1. module com.testModule{
2. exports com.prueba;
3. }

Con esto ya quedaría definido el módulo, como ves, no es muy diferente a crear un proyecto normal, lo único la existencia del archivo module-info.java con la descripción del módulo.

Para poder utilizar este módulo en otro módulo Java, dentro del archivo de definición del módulo cliente debemos incluir la instrucción ***requires***, seguida del nombre del módulo que queremos utilizar

1. module com.prueba.client {
2. requires com.testModule;
4. }

Esto permitirá utilizar en el nuevo proyecto el paquete de clases com.prueba del primer proyecto. Lo único que faltaría es añadir una referencia al proyecto origen en el proyecto destino a través de las propiedades del proyecto->Build Path pestaña *projects*:



**Apéndice B. Novedades Java 10**

Novedades en Java 10

Desde Java 9 la política de Oracle ha sido lanzar una nueva versión de Java cada seis meses, de ahí que las novedades en las últimas versiones sean más bien escasas respecto a la anterior.

En este documento vamos a describir algunas de las principales novedades que ha aparecido en la versión Java 10, en lo que a lenguaje y APIs de uso general se refiere.

Inferencia de tipos

A partir de Java 10, es posible declarar variables locales sin indicar el tipo de la misma, mediante el uso de la palabra reservada *var*:

var a=10;

A partir del valor asignado a la variable, el compilador "infiere" el tipo de la misma, en este caso int. La inferencia de tipos puede aplicarse a cualquier tipo Java, incluidas clases de cualquier tipo:

var datos=new ArrayList<Integer>();

Gracias a la inferencia de tipos, se evita tener que indicar nombres largos a la hora de definir el tipo de una variable cuando se trata de nombres de clases compuestos. Si se recomienda, a fin de claridad en el código, que el nombre de la variable sea lo suficientemente claro como para deducir de qué tipo de dato se trata:

var connection=DriverManager.getConnection(....);

A la hora de aplicar la inferencia de tipos, hay que tener en cuenta las siguientes consideraciones:

* Solo puede aplicarse con variables locales, nunca con atributos ni parámetros de métodos.
* La declaración y asignación de dato a la variable deben hacerse obligatoriamente en la misma instrucción, de manera que el compilador pueda determinar el tipo de la misma.
* Al tratarse de una operación en tiempo de compilación, la inferencia de tipo no empeora ni mejora el rendimiento de la aplicación en tiempo de ejecución.

Métodos de colección para creación de copias

Las interfaces de colección List, Map y Set proporcionan un método estático copyOf() para crear **copias no modificables** de una colección. En el caso de List y Set, este método recibe el objeto Collection que se quiere clonar:

·         List<E> copyOf(Collection<E> col) //método de List

·         Set<E> copyOf(Collection<E> col) //método de Set

Por su parte, en Map este método recibe como parámetro la implementación de Map que se quiere clonar:

·         Map<K,V> copyOf(Map<K,V> col) //método de Map

En el siguiente ejemplo vemos como crear una lista a partir de otra:

1. ArrayList<Integer> enteros=new ArrayList<>();
2. enteros.add(20);
3. enteros.add(10);
4. enteros.add(35);
5. //creación de copia no modificable
6. var copia=List.copyOf(enteros);
7. enteros.set(0, 4); //cambia la lista original, no la copia
8. //copia.set(0, 8); //esta operación no podría realizarse,
9. //provocaría una excepción UnsupportedOperationException
10. for(Integer i:copia) {
11. System.out.println(i);
12. }

Métodos de reducción de Collectors

La clase Collectors de Java 8 disponía de los métodos toList(), toSet() y toMap() para reducir un un stream a colección. Ahora en Java 10 se han añadido toUnmodificableList(), toUnmodificableSet() y toUnmodificableMap() para generar colecciones no modificables.

**Apéndice C. Novedades en Java 11**

Novedades en Java 11

En este documento vamos a describir algunas de las principales novedades que ha aparecido en la versión Java 11, en lo que a lenguaje y APIs de uso general se refiere.

Inferencia de tipos en parámetros de expresiones lambda

Java 10 incorporó la inferencia de tipos en variables locales. A partir de Java 11, también es posible aplicar esta características a los parámetros de una expresión lambda.

Por ejemplo, hasta Java 8 una expresión lambda que incluyera parámetros podría escribirse según estos ejemplos:

(int a, int b)->a\*b;

o simplemente:

(a,b)->a\*b;

A partir de Java 11, también puede escribirse de esta manera:

(var a, var b)->a\*b;

Si se decide utilizar var para inferir el tipo en los parámetros, deberá hacerse con todos ellos. Las siguientes expresiones lambda son incorrectas:

1. (var a,b)->a\*b;
2. (var a, int b)->a\*b;

Y, ¿Qué sentido tiene utilizar var para inferencia de tipos en parámetros de lambda si ya se puede inferir el tipo indicando solo el nombre de la variable?. Resulta que en aquellos casos en los que queramos anotar un parámetro de una expresión lambda solo podremos hacerlo si indicamos el tipo de la variable o bien usamos var. Por ejemplo, si queremos incluir la anotación @NonNull del checker framework para verificar que un parámetro no es null, deberíamos declarar el parámetro con var:

1. (@NotNull a)->System.out.println(a); //incorrecto
2. (@NotNull var a)->System.out.println(a); //correcto

Lanzamiento de programas de un único archivo de código fuente

Si un programa Java está formado por un único archivo de código fuente, ya no es necesario compilar el archivo desde línea de comandos con javac.exe y ejecutarlo después con java.exe, es posible utilizar el comando java.exe sobre el archivo de código fuente, lo que provocará la compilación y seguida ejecución del mismo:

>java Test.java  //compila primero y luego ejecuta la clase Test

Nuevos métodos en la clase String

Java 11 incorpora nuevos métodos en la clase String. Estos son:

boolean isBlank(). Devuelve *true* si la cadena está vacía o solamente contiene espacios en blanco. No debemos confundir con *isEmpty()*, que devuelve *true* solo si la longitud de la cadena es 0.

* **Stream lines()**. Devuelve un stream con las líneas que forman la cadena.
* **String repeat(int n)**. Devuelve una cadena resultado de concatenar la cadena actual, tantas veces como se indica en el parámetro.
* **String strip()**. Devuelve una nueva cadena resultante de eliminar espacios a izquierda y derecha. Es prácticamente igual *a trim()*, se diferencian en el concepto de espacio en blanco de cada uno, pues hay determinados código unicode que representan espacios en blanco pero *trim()*no los reconoce como tal, como por ejemplo '\u2000'.
* **String [stripLeading](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html" \l "stripLeading()" \t "_blank)()**.Devuelve una nueva cadena resultante de eliminar espacios a izquierda
* **String [stripTrailing](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html" \l "stripLeading()" \t "_blank)()**.Devuelve una nueva cadena resultante de eliminar espacios a derecha

Nuevos métodos de Files

La clase Files del paquete java.nio.file también incorpora nuevos e interesantes métodos en esta versión Java 11:

* **String readString(Path path)**. Devuelve un String con el contenido completo de un fichero, decodificando cada byte en un carácter según el estándar UTF-8. Existe otra versión de este método [readString](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html" \l "readString(java.nio.file.Path,java.nio.charset.Charset)" \t "_blank)​([Path](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Path.html" \o "interface in java.nio.file" \t "_blank) path, [Charset](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/charset/Charset.html" \o "class in java.nio.charset" \t "_blank) cs), que permite utilizar otro sistema de codificación.
* **Path writeString​(Path path, CharSequence csq, OpenOption... options)**. Escribe en el fichero la secuencia de caracteres especificada en el segundo parámetro, utilizando codificación UTF-8. La versión writeString​(Path path, CharSequence csq, Charset cs, OpenOption... options) permite utilizar otro sistema de codificación.

Mediante el siguiente código de ejemplo guardamos en un fichero el contenido de una cadena de caracteres:

1. String uri="c:\\temp\\datos.txt";
2. Path path=Paths.get(uri);
3. String cad="cadena de prueba para fichero";
4. Files.writeString(path, cad);