Novedades en *J2SE 5.0* (*Tiger*)

Iker Hurtado Díaz de Cerio

Autor

Sin experiencia en esto de escribir para las masas.

Motivado por las notables mejoras de la nueva versión, porque sobre este tema no abunda la información en castellano y por contribuir con esta nuestra comunidad.

Dedicatoria

Dedicado a mi prometida *forever* Raquel, que ha hecho y hace posible que me dedique a mi manera a lo que me gusta.

Agradecimientos a Emili Miedes de Elías y a Jesús Navarrete por sus correcciones y sugerencias.

Introducción

La nueva versión de *J2SE* trae cambios a todos los niveles: librerías, maquina virtual, despliegue, etc. Sin embargo las nuevas funcionalidades del lenguaje hacen que sea la actualización más importante de los últimos tiempos.

A pesar de los numerosos cambios que trae la nueva plataforma, la compatibilidad hacia atrás está garantizada. No es necesario reprogramar ni recompilar aplicaciones desarrolladas sobre versiones anteriores para que se ejecuten correctamente. Incluso se verán beneficiadas por las mejoras de ejecución que proporciona la nueva máquina virtual.

El documento se centra principalmente en las mejoras del lenguaje por su importancia y utilidad general. Para hacer justicia al título también se enumeran, sin entrar en explicaciones, el resto de novedades en la plataforma.

No todo lo expuesto en el documento forma parte de la especificación *J2SE 5.0*. También se incluyen mejoras específicas de la implementación de referencia y del kit de desarrollo *JDK 1.5* (de *Sun Microsystems*).

El contenido de este documento es adecuado para programadores de nivel medio, con conocimientos del lenguaje y la plataforma. No está redactado ni explicado con el detalle suficiente para personas noveles en *Java*.

Convenios utilizados

- No se hará uso o referencia a nuevas funcionalidades hasta que no se hayan estudiado.
- Los términos técnicos que no tienen una traducción clara o de mi gusto al castellano aparecerán en inglés.

- Glosario:
  - **tipo** engloba a clases e interfaces
  - **Colecciones** equivale a librería de colecciones o *Java Collections Framework*

- Los anglicismos o tecnicismos no traducidos aparecerán en cursiva.

- Los tipos del paquete *java.lang* serán referenciados sólo con el nombre del tipo.
  Ejemplo: *String* en vez de *java.lang.String*

- Representación del código:

```
Codigo correcto
```

```
Codigo con warnings
```

```
Codigo erroneo
```

Índice de contenido

1 Nuevas funcionalidades del lenguaje

1.1 Genéricos (*Generics*)

1.1.1 Introducción a los tipos genéricos

El soporte en el lenguaje de tipos y métodos genéricos es la característica más importante incorporada en la nueva versión de *Java*. Además se complementa muy bien con otras nuevas funcionalidades como *varargs*, anotaciones, tipos enumerados, etc.

Un tipo genérico permite controlar el tipo de los objetos con los que trabajan sus instancias cuando en su definición (del tipo genérico) no se quiere concretar (el tipo de los objetos) para hacerlo más general.

Un ejemplo claro de uso de genéricos es la librería de colecciones. Se usarán tipos de esta para los ejemplos de uso.

Hasta ahora (*J2SE 1.4*), esta necesidad de generalizar el tipo de los objetos con los que trabajan las instancias, se resolvía recurriendo al tipo raíz *Object*, lo que generaba problemas de seguridad de tipos y tareas molestas y repetitivas como *downcasting* o comprobaciones de tipos.
Usando genéricos la programación es más elegante y segura.

Vayamos con un ejemplo sencillo y típico para verlo más claro. Usaremos la interfaz *java.util.List* (a partir de ahora *List*).
El fin del tipo lista es almacenar una lista de elementos (que soporte elementos de cualquier tipo).

Hasta ahora (*J2SE 1.4*), para lograrlo se usaba (en la definición de *List*) el tipo *Object* (tipo raíz) como tipo de los elementos de la lista, consiguiendo soporte para cualquier tipo, ya que todos descienden de *Object*.

El problema de esta solución es que si quieres que la lista sólo mantenga elementos de un tipo (que es lo más frecuente), como por ejemplo *String*, el compilador no te lo garantiza.

Con los tipos genéricos sin embargo, y gracias a las variables de tipo (*type variable*), es posible instanciar una lista de *Strings*, y el compilador controlará el tipo de los elementos y facilitará el trabajo con ellos.
Las variables de tipo tienen la función de ser sustituidas por tipos concretos cuando se crea una instancia del tipo genérico en cuestión.

Ejemplo de firma de un tipo genérico

Para entender mejor lo anterior me parece importante poner un ejemplo sencillo de definición de tipo genérico. Seguimos con *List*, definido de la siguiente manera:

```
public interface List<E> extends Collection<E>
```

Como se puede observar en la definición, el tipo *List* es parametrizable, tiene como parámetro la variable de tipo *E*.

Ejemplo de instanciación de un tipo genérico

```
List<String> listaDeStrings= new LinkedList<String>();
```

En este caso, el tipo genérico *List* es instanciado parametrizado con el tipo String, con lo que se le dice al compilador que los elementos de la lista queremos que sean del tipo *String*.

1.1.2 Uso de tipos genéricos

En este apartado se presentan ejemplos básicos de uso de genéricos.

Cualquiera que haya trabajado con colecciones ha echado de menos los genéricos. En *J2SE5.0* se ha reescrito la librería para beneficiarse de las ventajas de los genéricos,. Así que se usarán tipos de la librería de colecciones para los ejemplos de código.

Ejemplos sencillos que comparan la forma de trabajar hasta ahora con el trabajo con genéricos.

Hasta ahora:

```
List lista= new LinkedList();
lista.add("Hola");
String elemento= (String)lista.get(0);
```

Usando genéricos:

```
List<String> listaDeStrings= new LinkedList<String>();
listaDeStrings.add("Hola");
String elemento= listaDeStrings.get(0);
```

Como se puede ver, parametrizamos la lista en su instanciación con el tipo String. De esta forma el compilador controla que los elementos añadidos a la lista sean del tipo *String* y a la hora de su extracción ya no es necesario hacer el *downcasting*, el compilador lo hace por nosotros automáticamente.

Otro ejemplo, en este caso con la interfaz *java.util.Map* parametrizada con dos variables de tipo, una para el tipo de la clave y la otra para el tipo del valor de los elementos (*Interface Map<K,V>*).

Hasta ahora:

```
Map mapa= new HashMap();
mapa.put("uno",new Integer(1));
mapa.put("dos",new Integer(2));
Integer entero= (Integer)mapa.get("uno");
```

Usando genéricos:

```
Map<String,Integer> mapaStringInteger= new
HashMap<String,Integer>();
mapaStringInteger.put("uno",new Integer(1));
mapaStringInteger.put("dos",new Integer(2));
Integer entero= mapaStringInteger.get("uno");
```

Como se puede deducir, en este caso se controlan los tipos de la clave y del valor de los elementos.

Vamos con otro ejemplo que mostrará la potencia de los genéricos para definir y manejar estructuras de datos complejas, gracias a la posibilidad de usar tipos genéricos como parámetros de otros tipos genéricos:

```
Map<String,List<Map<String,int[]>>> estructuraCompleja = new
HashMap<String,List<Map<String,int[]>>>();

int i= estructuraCompleja.get(clave1).get(0).get(clave2)[0];
```

Es especialmente notable la simplificación en la obtención de información de la estructura, que sin el uso de genéricos tendría una notación mucho más engorrosa:

```
int i= ((int[])((Map)((List)estructuraCompleja
.get(clave1)).get(0)).get(clave2))[0];
```

1.1.3 Aspectos avanzados del uso de tipos genéricos

En este apartado abordaremos una serie de detalles y cuestiones que se plantean con el uso de genéricos.

1.1.3.1 Raw types

Aunque, como se ha visto anteriormente, los tipos de la librería de colecciones se han hecho genéricos en *J2SE 5.0*. También es posible instanciarlos como hasta ahora, sin pasar un tipo como parámetro y se les conoce como *raw types*.
Con esto se consigue compatibilidad hacia atrás, el código programado para *J2SE 1.4* es aceptado, aunque el compilador nos avisará de que esta práctica no es recomendable:

```
List lista= new LinkedList();
lista.add("Hola");
lista.add(new Integer(5));
```

En este caso el compilador se queja en el método *add(),* ya que no puede garantizar la seguridad de tipos al añadir los elementos.

No hay mucha justificación para utilizar *raw types* en código nuevo. Si lo que se quiere es crear una lista que acepte cualquier tipo de objeto, con *J2SE 5.0,* lo propio sería:

```
List<Object> lista = new LinkedList<Object>( );
lista.add("Hola");
lista.add(new Integer(5));
```

1.1.3.2 Jerarquía de tipos genéricos

La jerarquía de los tipos genéricos es independiente de la jerarquía de sus tipos parametrizados.

Mejor explicarlo con ejemplos:

```
Collection<String> coleccionDeStrings= new LinkedList<String>();
```

Este es el caso correcto, una lista de *Strings* es una colección de *Strings*.

```
List<Object> listaDeObjects= new LinkedList<String>();
```

Esta asignación es incorrecta, error de compilación. No hay ninguna relación jerárquica entre los dos objetos, a pesar de la relación entre los tipos parametrizados (*Object* y *String*). Si esto se permitiera se podrían añadir a la lista elementos de tipo diferente a *String* perdiéndose la seguridad de tipos.

```
Collection<Object> coleccionDeStrings= new LinkedList<String>();
```

Incorrecto, una lista de *Strings* no es una colección de *Objects*, realmente es el mismo caso que el anterior.

1.1.3.3 (In)Seguridad de tipos en tiempo de ejecución

Como consecuencia del soporte de *raw types*, es posible generar un error de tipos en tiempo de ejecución (*ClassCastException*).
El agujero de seguridad de tipos se produce por la posibilidad de convertir un tipo parametrizado en no parametrizado. Ejemplo:

```
List<String> listaDeStrings= new ArrayList<String>();
List lista= listaDeStrings;
lista.add(new Integer(1));
String s0= listaDeStrings.get(0);  // ClassCastException en ejecución
```

El compilador avisa pero lo permite, es el precio que hay que pagar por la compatibilidad hacia atrás.

De todas formas para casos necesarios existen en la clase *java.util.Collections* varios métodos (*chequedList,chequedMap*...) que comprueban la seguridad de tipos en tiempo de ejecución en los ejemplares que devuelven.

1.1.3.4 Arrays de tipos parametrizados

En *Java*, un *array* de un tipo es también un *array* de una superclase o interfaz de ese tipo. Así, el intérprete tiene que analizar en tiempo de ejecución el tipo de los objetos que se intentan introducir.

```
String[] palabras = new String[10];
Object[] objetos = palabras;
objetos[0] = new Integer(1);     // lanza una ArrayStoreException
```

En la ejecución el ejemplo anterior, el intérprete sabe que *objetos* es un *array* de *String* y al comprobar que el objeto a asignar es de otro tipo (*Integer*) lanza una excepción.

Con los tipos genéricos esta protección en tiempo de ejecución no es suficiente, ya que el control de tipos se efectúa en tiempo de compilación. La consecuencia sería otro agujero en el control de tipos. Para evitarlo el compilador no permite declaraciones de *arrays* de tipos parametrizados:

```
List<String>[] listaDeStrings = new ArrayList<String>[10];  // Error de compilacion
```

aunque sí de tipos genéricos sin parametrizar (por compatibilidad hacia atrás), de esta desaconsejable forma:

```
List[] lista= new ArrayList[10];
```

1.1.3.5 Comodín de tipo (Type Parameter Wildcard)

Cuando se necesita que un método cualquiera (no tiene que ser genérico) admita cualquier instancia de un tipo genérico (parametrizada con cualquier tipo) como parámetro, es posible usar el comodín de tipo o tipo desconocido. Se representa con el carácter ?, en vez de un tipo concreto.

Como ejemplo el método *reverse* de la clase *java.util.Collections*

```
static void reverse(List<?> lista)
```

Con el comodín de tipo se logra que el método invierta el orden listas parametrizadas con cualquier tipo.

Se podría conseguir el mismo resultado declarando como parámetro el *raw type* de lista:

```
static void reverse(List lista)
```

Pero como ya se ha dicho, esto se permite por compatibilidad hacia atrás y esta totalmente desaconsejado.

En la definición de estos métodos, el uso del comodín de tipo tiene dos implicaciones importantes a tener en cuenta, debidas al desconocimiento del tipo:

1. Los métodos del tipo genérico (*List<E>*) que están declarados para devolver un valor del mismo tipo que el parámetro de tipo (*E*), en las condiciones de desconocimiento de tipo devuelven un *Object*.

   Ej: *E get(int index)* devolverá *Object* en vez de *E*

2. Los métodos del tipo genérico (*List<E>*) que están declarados para aceptar argumentos cuyo tipo es el parámetro de tipo (*E*), en las condiciones de desconocimiento de tipo no pueden ser llamados. El compilador no lo permite porque no garantizaría la seguridad de tipos.

   Ej: *boolean add(E o)* no podrá ser llamado

Comodín de tipo limitado

Es posible usar el concepto de comodín de tipo pero limitándolo a determinadas clases, las que descienden o son superclases de una dada.

Véase con un ejemplo: necesitamos un método que sume una lista de números de cualquier tipo.

La clase *Number* es superclase de todas las que representan números.

Por lo visto hasta ahora no podríamos usar como parámetro *java.util.List<Number>* ya que nos restringiría demasiado. No sería posible sumar listas de tipos descendientes de *Number*, por ejemplo *java.util.List<Integer>*.

La otra opción sería usar *List<?>*, pero en ese caso los elementos serían tratados como *Object* por lo que no se podrían operar matemáticamente.

El comodín de tipo limitado nos permite una solución más adecuada para este ejemplo:

```
public static double sumaLista(List<? extends Number>
listaNumerica){
      double total = 0.0;
      for (int i= 0; i < listaNumerica.size(); i++){
            total += listaNumerica.get(i).doubleValue();
      }

      return total;
}
```

Así, sólo se permiten listas parametrizadas con clases derivadas de *Number* (incluido la propia *Number*).

Respecto a las dos implicaciones que apuntamos para el comodín de tipo normal, aquí se aplican análogamente:
1. Se devuelve el tipo de la superclase limitante en vez de *Object*, en el ejemplo *Number*.
2. Lo mismo, no se permite la llamada de estos métodos.

En el ejemplo anterior hay una limitación superior (*<? extends Superclase>*) pero también es posible limitar inferiormente (*<? super Subclase>*), en este caso las clases posibles serán superclases de la dada.

Respecto a las dos implicaciones que apuntamos del uso de comodín de tipo, en este caso serían:
1. Se devuelve el tipo *Object*.
2. Es seguro llamar a los métodos pasando como argumento el tipo especificado como subclase.

1.1.4 Definición de tipos genéricos

Formalmente los tipos genéricos son tipos que tienen como parámetros variables de tipo (*type variables*) y uno o más métodos que las utilizan como argumentos o valor de retorno.

Empezamos con un ejemplo sencillo, la definición de una clase genérica que modela una estructura de tipo pila (*LIFO - Last Input First Output*).

```
            Implementación sencilla de Pila

public class Pila<E> {

      List<E> pila= new LinkedList<E>();
```

```
    void aniadir(E elemento){

        pila.add(elemento);
    }


    E extraer(){

        int i= pila.size()-1;
        E elemento;
        if (i < 0){
            elemento= null;
        }else{
            elemento= pila.get(i);
            pila.remove(i);
        }

        return elemento;
    }


    boolean estaVacia(){

        return pila.isEmpty();
    }

}
```

El hecho de ser genérica confiere a la clase el control del tipo de sus elementos.

La definición de una clase genérica requiere variable o variables de tipo que posibiliten la parametrización del tipo. Las variables de tipo se declaran entre los símbolos < y > y separadas por comas, y se sitúan después del nombre del tipo a definir. Por convención, y para no confundirse con otros identificadores, el nombre de las variables de tipo suele ser una letra mayúscula.

Forma general:

NombreClaseGenerica<A,B,C,...>

A la hora de usar las variables de tipo hay que tener en cuenta que no existen en tiempo de ejecución por lo que no se pueden usar con operadores como *instanceof* y *new*.

Las variables de tipo no tienen sentido en los campos estáticos, ya que no son instanciables y por lo tanto parametrizables.

Variables de tipo limitadas

En la definición de tipos genéricos es posible limitar las variables de tipo, restringiendo sus valores a tipos que implementan una o más interfaces y/o que son subclases de una dada.

Se ha ampliado el ejemplo anterior restringiendo la variable de tipo *E* a tipos que hereden de *Number* y que implementen la interfaz *Comparable*, parametrizada con ese mismo tipo, con el fin de poder implementar la interfaz *Comparable* para el tipo genérico:

Implementación comparable de Pila

```
public class Pila<E  extends Number & Comparable<E>> implements
Comparable<Pila<E>>{

     List<E> pila= new LinkedList<E>();


     void aniadir(E elemento){

          pila.add(elemento);
     }


     E extraer(){

          int i= pila.size()-1;
          E elemento;

          if (i < 0){
               elemento= null;
          }else{
               elemento= pila.get(i);
               pila.remove(i);
          }

          return elemento;
     }


     boolean estaVacia(){

          return pila.isEmpty();
     }


     public int compareTo(Pila<E> otraPila) {

          if (estaVacia() && otraPila.estaVacia()) return 0;
          if (estaVacia()) return -1;
          if (otraPila.estaVacia()) return 1;

          int result;
          E elto1= extraer();
          E elto2= otraPila.extraer();

          if (elto1.compareTo(elto2)==0)
               result= compareTo(otraPila);
          else result= elto1.compareTo(elto2);

          otraPila.aniadir(elto2);
          aniadir(elto1);

          return result;
    }

 }
```

La notación que se utiliza para limitar una variable de tipo consiste en seguir el nombre de la variable de *extends* seguido de las interfaces (separadas por el símbolo *ampersand*) y/o la superclase. Forma general:

<T ***extends*** *Clase* ***&*** *Intefaz1* ***&*** *Interfaz2* ***&*** *....>*

1.1.5 Métodos genéricos

Los métodos genéricos son métodos que tienen declaradas sus propias variables de tipo.

El ejemplo típico son los métodos estáticos, que por su naturaleza estática, no pueden acceder a las variables de tipo declaradas en su clase, pero sí pueden tener sus propias variables de tipo.

Definición de métodos genéricos

Las variables se declaran como hasta ahora, entre los símbolos < y > y separadas por comas. Se sitúan entre los modificadores del método y el tipo de retorno (se verá en los ejemplos), forma general:

modificadoresMetodo <A,B,..> TipoRetorno nombreMetodo(arg1,arg2,..)

¿Qué utilidad pueden tener los métodos genéricos? Se podrían usar para limitar los tipos parametrizables del tipo genérico. Lo vemos con el ejemplo de la versión sencilla de la `Pila`:

Podríamos añadir a la clase `Pila` un método estático que devolviera el sumatorio de los elementos de la pila, con lo que hay que limitar como argumentos pilas parametrizadas con tipos derivados de *Number*:

```
public static <N extends Number> double sumatorio(Pila<N> p){

     double total= 0;
     for (int i = 0; i < p.pila.size(); i++)
          total+= p.pila.get(i).doubleValue();

     return total;
}
```

Como puede verse se utiliza la variable de tipo *N*, para limitar las instancias de `Pila`.

Esto se podría conseguir más adecuadamente, usando comodines de tipo (ya estudiados):

```
public static double sumatorio(Pila<? extends Number> p){

     double total= 0;
     for (int i = 0; i < p.pila.size(); i++)
               total+= p.pila.get(i).doubleValue();

     return total;
}
```

La utilidad real de los métodos genéricos es establecer relaciones entre los tipos de los parámetros o entre tipos de los parámetros y el tipo de retorno en un método. Siguiendo con el ejemplo de la pila, podríamos necesitar un método que nos devolviera la pila con mayor sumatorio de sus elementos:

```
public static <N extends Number> Pila<N> mayorSumatorio(Pila<N> p1,
Pila<N> p2) {

     Pila<N> pilaResultado= null;

     if (sumatorio(p1) > sumatorio(p2)) pilaResultado= p1;
     else if (sumatorio(p1) < sumatorio(p2)) pilaResultado= p2;

     return pilaResultado;
}
```

En la definición de este método además de restringir el tipo parametrizable a descendiente de *Number*, relaciona los tipos de los dos argumentos y del retorno. La relación en este caso es de igualdad.

Así, el método sólo permitirá comparar, por ejemplo, pilas de enteros pero no una pila de enteros con una parametrizada con *Double*, aunque sea descendiente de *Number*. Por supuesto el resultado sería una pila de enteros.

Uso de métodos genéricos

Al contrario que los tipos genéricos, en la llamada a un método genérico no suele ser necesario especificar el valor de la variable de tipo, el compilador se encarga de deducirlo a partir de los argumentos del método. Seguimos con el ejemplo definido anteriormente:

```
Pila<Double> p1=  new Pila<Double>();
p1.aniadir(111.2);

Pila<Double> p2=  new Pila<Double>();
p2.aniadir(2.2);

Pila<Double> p3=  Pila.mayorSumatorio(p,p2);
```

El compilador deduce que el tipo parametrizado es *Double*, por lo que no es necesario ponerlo, aunque tampoco sería incorrecto:

```
Pila<Double> p3=  Pila.<Double>mayorSumatorio(p,p2);
```

Hay muchos métodos genéricos para tomar como ejemplos en la clase *java.util.Collections* del *framework* de colecciones.

1.1.6 Excepciones parametrizadas

Las excepciones son clases que se generan y capturan en tiempo de ejecución, por lo que es imposible para el compilador controlar los tipos de excepciones parametrizadas. Por este motivo no

es posible introducir variables ni comodines de tipo en los bloques *catch*.

En conclusión, no es posible hacer genérico ningún tipo derivado de *Throwable*, por lo que las excepciones parametrizadas no están permitidas.

Para profundizar más y aprender a usar de la mejor manera las posibilidades que ofrecen los genéricos, lo mejor es consultar las implementaciones de la librería estándar de *Java5. S*on usados ampliamente en los paquetes *java.util, java.lang, java.lang.reflect* y *java.util.concurrent.*

1.2 Tipos enumerados

Los tipos enumerados son una nueva "especie" de tipos (como los tipos clase y los tipos interfaz) del lenguaje que se caracterizan por tener un número finito y normalmente bajo de posibles valores.

Los enumerados implementan el concepto de constante, de uso frecuente en programación, que hasta ahora se implementaba con variables *static final int*. La formalización que proporcionan los tipos enumerados lleva a otro nivel de seguridad y funcionalidad el uso de constantes en *Java*.

1.2.1 Definición de tipos enumerados simples

La palabra reservada para la definición de tipos enumerados es *enum.* Irá seguida del nombre del tipo y del cuerpo de la declaración donde se listarán los posibles valores separados por comas. Veamos un ejemplo sencillo:

```
public enum NotasMusicales{ DO,RE,MI,FA,SOL,LA,SI }
```

Forma general: *modificadores* ***enum*** *NombreTipoEnumerado{ valor1,valor2,.. }*

1.2.2 Valores enumerados

Los valores enumerados son los valores que puede tomar el tipo enumerado. Conceptualmente son constantes del lenguaje. Normalmente se definen con letras mayúsculas y se accede a ellos de esta forma:

NombreTipoEnum.NOMBRE_VALOR

Ejemplo:

```
NotasMusicales notaActual= NotasMusicales.FA;
```

1.2.3 Particularidades de los tipos enumerados

Los tipos enumerados en el fondo son clases. Unas clases muy especiales y con un comportamiento limitado en algunos aspectos. Algunas particularidades importantes de los tipos enumerados son:

- Las únicas instancias de un tipo enumerado son las que se crean en su definición. Consecuencias:
  - Los tipos enumerados no tienen constructor público.
  - Las instancias no son clonables. No implementan la interfaz *Cloneable*.

    - Las instancias son inmutables.
    - Es seguro comparar instancias con el comparador ==.

      Es más el método *equals()* está implementado con == y es *final* por lo que no puede ser modificado.
- Los valores enumerados de un tipo son ordinales, mantienen el orden de su declaración en la definición del tipo. Consecuencias:
    - Los tipos enumerados implementan la interfaz *Comparable*.
    - Los tipos enumerados implementan el método *ordinal()* (es *final*) que devuelve un entero reflejando la posición del valor. El primer valor tiene la posición 0. No suele ser muy utilizado.
    - Los tipos enumerados implementan el método *values()* (es *final*) que devuelve un *array* con todos los valores del tipo ordenados.
- Los tipos enumerados implementan el método *toString()* que devuelve el nombre del valor. No es *final* y podría reescribirse. También implementa el método estático *valueOf()* que hace lo contrario.
- Los tipos enumerados implementan la interfaz *Serializable*, aunque cuidando de que no se creen nuevas instancias.
- Los tipos enumerados son subclases de *Enum* (nuevo en *Java 5.0*), pero no es posible definir un tipo enumerado manualmente heredando de *Enum* (error de compilación). *Enum* no es un tipo enumerado.
- No es posible extender un tipo enumerado, ya que es *final* por definición (no es necesario especificarlo).
- Los tipos enumerados pueden implementar interfaces.

1.2.4 Soporte de *switch* a tipos enumerados

Hasta *J2SE 5.0* la sentencia *switch* sólo soportaba los tipos primitivos *int*, *short*, *char* y byte. Ahora se ha extendido el soporte a los tipos genéricos, que por otra parte se adaptan perfectamente a la sentencia debido a su limitado número de valores posibles.

Veamos un ejemplo:

```
switch(notaActual) {
      case DO:
            ...
            break;
      case RE:
            .....
            break;
      default:
            ....
}
```

Apuntes:

- Como se puede ver en el ejemplo, los valores tratados en la sentencia no están precedidos del nombre del tipo. Es evidente que no es necesario (el compilador conoce el tipo), pero lo sorprendente es que no es posible ponerlo, daría un error de compilación.

- Si la variable enumerada que se evalúa en la sentencia es *null*, se lanza una *NullPointerExcepcion*. Tampoco puede ponerse *null* como un valor tratado.

- El compilador controla si la sentencia está tratando todos los valores posibles de tipo enumerado y en caso de no hacerlo, reporta un aviso en la compilación.

1.2.5 *EnumMap* y *EnumSet*

EnumMap y EnumSet son implementaciones de las interfaces *Map* y *Set* que aprovechan la particularidad de los tipos enumerados para conseguir más eficiencia y compactación.

EnumMap

EnumMap es una implementación de la interfaz *Map* donde el tipo de las claves del mapa es un enumerado. La implementación aprovecha el conocimiento del número de valores posibles del tipo enumerado, resultando muy rápida y compacta.

Ejemplo de creación:

```
Map<NotasMusicales,String> mapaNotas= new
EnumMap<NotasMusicales,String>(NotasMusicales.class);
```

EnumSet

EnumSet es una clase que implementa la interfaz *Set* donde el tipo de los elementos del conjunto es un tipo enumerado. La implementación aprovecha el conocimiento sobre el tipo enumerado, para representar internamente el conjunto como un vector de bits, consiguiendo una gran eficiencia.

Además de implementar la interfaz *Set*, la clase define una larga serie de "constructores" estáticos, que dan una gran flexibilidad a la hora de crear conjuntos.

Ejemplo de creación:

```
Set<NotasMusicales> conjNotas= EnumSet.allOf(NotasMusicales.class);
```

1.2.6 Sintaxis avanzada de tipos enumerados

Hasta ahora se ha visto la definición de tipos enumerados en versión sencilla, pero la sintaxis de los tipos enumerados es más potente y compleja:

- Se pueden definir campos, métodos y constructores en un tipo enumerado.
- Si se define uno o más constructores, es posible llamarlos siguiendo los valores enumerados de los argumentos correspondientes al constructor.
- Cada valor enumerado puede tener su propio cuerpo de clase donde es posible sobreescribir métodos del tipo.

Vamos con un ejemplo:

```
public enum FormatoAudio {
      PCM("2"),
      MP3("2",192),
      DOLBYDIGITAL("5.1",640);

      private String canales;
      private int bitrate;

      FormatoAudio(String canales,int bitrate) {
            this.canales = canales;
            this.bitrate = bitrate;
      }

      FormatoAudio(String canales) {
            this.canales = canales;
            this.bitrate = 0;
      }


      public int getBitrate() {
            return this.bitrate;
      }

      public String getCanales() {
            return this.canales;
      }

      public String descripcion() {
            return "El formato de audio "+this+" tiene
"+this.canales+" canales y un bitrate de "+this.bitrate+" kbps.";
      }

}
```

Como se puede ver, lo primero que se declara, igual que en el caso sencillo, son los valores enumerados. Aunque aquí están parametrizados ajustándose a uno de los dos constructores.

Le sigue (separados por ;) el cuerpo del tipo, compuesto por las variables de instancia, donde se puede guardar más información sobre cada valor enumerado. Por último los métodos que aportan funcionalidad.

Realmente lo nuevo es igual que en las clases normales, no tiene más misterio.

Las particularidades descritas en un apartado anterior son perfectamente aplicables a los enumerados complejos.

Como se ha dicho al inicio del apartado, cada valor enumerado puede tener su propio cuerpo de clase logrando un comportamiento específico del valor, que puede ser útil en algunas ocasiones.

Definir un cuerpo de clase para un valor enumerado es sencillo, se coloca el cuerpo entre llaves después del valor.

Vamos con el ejemplo de FormatAudio:

```
public enum FormatoAudio {
      PCM("2"){
            public void reproducir(String nombreFichero) {
                  ......
            }
      },
      MP3("2",192){
            public void reproducir(String nombreFichero) {
                  .....
            }
      },
      DOLBYDIGITAL("5.1",640){
            public void reproducir(String nombreFichero) {
                  ....
            }
      };

      private String canales;
      private int bitrate;

      FormatoAudio(String canales,int bitrate) {
            this.canales = canales;
            this.bitrate = bitrate;
      }

      FormatoAudio(String canales) {
            this.canales = canales;
            this.bitrate = 0;
      }


      public int getBitrate() {
            return this.bitrate;
      }

      public String getCanales() {
            return this.canales;
      }

      public String descripcion() {
            return "El formato de audio "+this+" tiene
"+this.canales+" canales y un bitrate de "+this.bitrate+" kbps.";
      }


      public abstract void reproducir(String nombreFichero);

}
```

En el ejemplo se define un nuevo método abstracto en el tipo; *reproducir()*, que tiene que ser implementado (no se implementan en el ejemplo por abreviar) en cada instancia, o sea en cada valor enumerado.

Al redefinir un método se está creando para cada valor una subclase anónima del tipo enumerado.

De esto se deduce que los tipos enumerados aunque no pueden ser extendidos, no son estrictamente *final*, ya que pueden tener subclases anónimas.

1.3 Anotaciones

1.3.1 Introducción

Las anotaciones permiten asociar información, en forma de pares atributo-valor (miembros), a los elementos de programación (paquetes, tipos, métodos, constructores, campos, parámetros y variables locales). Las anotaciones son meta información, información sobre el código(información).

Un tipo de anotación (*annotation type*) es a una anotación lo que a un objeto es su clase. Se define como una interfaz especial con restricciones y diferente sintaxis, se verá su definición más adelante.

Las anotaciones no modifican la ejecución de un programa, por lo que sólo tienen sentido para programas o herramientas que las extraigan e interpreten. Un ejemplo es el compilador, que procesa varios tipos de anotaciones predefinidas. El *JDK* proporciona un *framework* para herramientas de procesado de anotaciones llamado *apt*.

La nueva *Java Reflection API* de *J2SE 5.0* soporta anotaciones por lo que se puede tener acceso a las anotaciones en tiempo de ejecución.

Los miembros de un tipo de anotación son los pares atributo-valor que soporta. Una anotación marcador (*marker annotation*) es un tipo de anotación que no declara ningún miembro, la información que aporta al elemento es su presencia o ausencia.

Todo lo dicho se extenderá y aclarará en los siguientes apartados.

1.3.2 Uso de anotaciones

Sintaxis

La sintaxis tiene esta forma general:

@TipoAnotacion(nombre1=valor1,nombre2=valor2,...)

El orden de los pares no es importante. Si un miembro tiene valor por defecto no es obligatoria su presencia.

Si el tipo de anotación es marcador, no tiene miembros, no son necesarios los paréntesis. Quedaría:

@TipoAnotacion

Si el tipo tiene un único miembro y tiene de nombre *value*, podría suprimirse el nombre y el símbolo =. Quedaría la forma general:

@TipoAnotacion(valor)

Si el tipo de un miembro es un *array* de elementos, se pasan los valores entre llaves y separados por comas. Forma general:

@TipoAnotacion(....,nombren={subvalor1,subvalor2,...},...)

Si además sólo se quiere pasar un valor en el *array* no son necesarias las llaves. Forma general:

@TipoAnotacion(....,nombren=subvalor,...)

Los valores tienen que coincidir con el tipo del miembro declarado en su definición.

Colocación

Las anotaciones por norma general se colocan justo antes de los modificadores de los elementos del lenguaje. Normalmente en la línea anterior a la del elemento.

Un caso especial son los paquetes, que como no son declarados en ningún sitio, es necesario para asignarles una anotación, crear un fichero llamado *package-info.java* donde se declara el paquete y se antepone la anotación.

También hay que apuntar que los valores enumerados, a pesar de no tener modificadores, pueden ser anotados normalmente.

Para terminar un elemento no puede tener más de una instancia de un tipo de anotación.

1.3.3 Anotaciones estándar

En *J2SE 5.0* hay tres anotaciones estándar que forman parte del paquete java.lang. Servirán como primeros ejemplos. Como se ha dicho, las anotaciones son entendidas por alguna herramienta, en el caso que nos ocupa es el compilador el que utiliza las tres anotaciones.

1.3.3.1 Override

Es un tipo de anotación que tiene como objetivo avisar al compilador que el método al que acompaña sobreescribe un método de la superclase. El compilador con esta información lo comprueba y en caso de no ser cierto, genera un error de compilación. Así se consigue evitar errores de programación peligrosos, como equivocarse en el nombre del método a sobreescribir.

Override es una anotación marcador y sólo se puede aplicar a métodos, no a otros elementos.

Ejemplo de sobreescritura de *toString()*

```
@Override
public String toString() {
      ....
}
```

1.3.3.2 Deprecated

Tipo de anotación que avisa al compilador de que un elemento es obsoleto, que se desaconseja su uso. De la misma forma que lo hace la etiqueta *@deprecated* a la herramienta *javadoc*. El compilador lanzará un *warning*, cuando se esté usando un elemento obsoleto en código no obsoleto.

Deprecated es una anotación marcador y se puede aplicar a cualquier elemento.

Ejemplo de clase obsoleta:

```
@Deprecated
public enum FormatoAudio {..
```

1.3.3.3 SuppressWarnings

Tipo de anotación que indica al compilador qué tipo de *warnings* no debe lanzar de las generadas en el elemento anotado.

Se utiliza cuando el programador es consciente de que esta haciendo una práctica desaconsejada pero por circunstancias es necesario, de está forma en una zona determinada del código se deshabilita al compilador para generar *warnings* de un tipo.

SuppressWarnings tiene un miembro cuyo tipo es un *array* de *Strings*, que contendrá los identificadores de los *warnings* que se quieren evitar. Actualmente los identificadores no están especificados por lo que se depende de la implementación del compilador. Algunos implementados son: *all, deprecation, checked, fallthrough, path, serial, finally*.

Cuando un tipo de anotación sólo define un miembro, como es el caso, sólo es necesario declarar el valor, que en este caso es un *arrray*. Vemos un ejemplo:

```
@SuppressWarnings({"checked","finally"})
public enum FormatoAudio {..
```

Incluso si el *array* que se quiere pasar tiene sólo un elemento, no son necesarias las llaves:

```
@SuppressWarnings({"all"})
public enum FormatoAudio {..
```

Para terminar, decir que *SuppressWarnings* se puede aplicar a casi todos los elementos: tipos, campos, métodos, parámetros, constructores y variables locales.

1.3.4 Definición de tipos de anotación

Un tipo de anotación es una interfaz especial. Esto se refleja en su definición que puede resultar bastante extraña. Estas son sus particularidades:

- Un tipo de anotación se define con la palabra clave *@interface*, que implícitamente extiende la interfaz *java.lang.annotation.Annotation*. Si se extiende manualmente no se crea un tipo de anotación.
- Los miembros del tipo se definen como métodos sin parámetros, donde:
  - El nombre del miembro será el nombre del método.
  - El tipo del miembro es el tipo de retorno del método, y puede ser: un tipo primitivo, *String*, *Class*, un tipo enumerado, un tipo de anotación o un *array* simple de un tipo de los anteriores.
  - El valor por defecto del miembro se define siguiendo el método de la palabra *default* seguida del valor correspondiente, que evidentemente será compatible con el tipo del miembro. *null*

no es un valor por defecto válido.

  - El único tipo de miembro que puede ser parametrizado es *Class*.
- Como cualquier interfaz, un tipo de anotación puede definir constantes y miembros estáticos y puede ser implementada o extendida.
- Un tipo de anotación puede ser anotada en su definición por meta-anotaciones, que se explican en el apartado siguiente.

1.3.4.1 Meta-anotaciones

Las meta-anotaciones son anotaciones aplicables a anotaciones. *Java 5.0* define cuatro meta-anotaciones estándar (paquete *java.lang.annotation*) que se pueden usar en la definición de anotaciones para caracterizarlas de alguna manera. Son:

Target

Meta-anotación que especifica los elementos "objetivo" del tipo de anotación que está siendo definida. O sea los elementos del lenguaje que pueden ser anotados por el tipo.

Si un tipo de anotación no tiene la meta-anotación *Target*, esta puede ser usada en cualquier elemento del lenguaje.

Target tiene un único miembro de nombre *value* y de tipo *java.lang.annotation.ElementType[]* , un *array* de un tipo enumerado que representa los elementos del lenguaje.

Retention

Retention especifica el alcance del tipo de anotación que está siendo definida. Tiene un único miembro llamado *value*, de tipo *java.lang.annotation.RetentionPolicy* con estos valores posibles:

- *SOURCE* La anotación sólo existe en el código fuente, no pasa a la clase compilada.
- *CLASS* La anotación se registra en la clase, pero la máquina virtual la ignora en tiempo de ejecución.
- *RUNTIME* Además de quedar registrada en la clase, la máquina virtual la retiene en tiempo de ejecución siendo accesible reflexivamente.

El valor por defecto es *CLASS.*

Nota: El formato de fichero *class* no es capaz de almacenar anotaciones para las variables locales. Por lo tanto aunque una variable local este acompañada por una anotación de alcance *CLASS* o *RUNTIME*, no será efectivo, quedándose siempre en *SOURCE*.

Documented

Especifica que el tipo de anotación definida debe formar parte de la documentación generada por herramientas como *javadoc*.

Documented es una anotación marcador, no tiene miembros.

Inherited

Inherited es una anotación marcador que especifica que el tipo de anotación definido es heredado. Quiere decir que la anotación aplicada a una clase se aplicará también a sus subclases.

1.3.4.2 Ejemplos

La forma general de la definición de tipos de anotaciones podría ser:

@Meta-anotacion1(miembros)
@Meta-anotacion2(miembros)
.....
modificadores ***@interface*** *NombreTipoAnotación{*
    TipoMiembro1 nombreMiembro1();
    TipoMiembro2 nombreMiembro2() ***default*** *valorPorDefecto;*
    ...
}

Ejemplo sencillo: definición de la anotación *java.lang.Override* en *J2SE 5.0*:

```
package java.lang;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

Otro ejemplo: definición de la anotación *java.lang.SuppressWarnings* en *J2SE 5.0*:

```
package java.lang;

import java.lang.annotation.*;
import java.lang.annotation.ElementType;
import static java.lang.annotation.ElementType.*;


@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR,
LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
       String[] value();
}
```

En este caso, como ya se dijo en la descripción de esta anotación estándar, se define un miembro de tipo *array* de *Strings* llamado *value*.

1.3.4.3 Anotaciones y Reflection API

La *Reflection API* de *J2SE 5.0* ha sido modificada para soportar anotaciones. Eso sí, sólo las visibles en tiempo de ejecución, las definidas con alcance (*retention*) *RUNTIME*.

La interfaz que permite leer reflexivamente las anotaciones es *java.lang.reflect.AnnotatedElement*. Es implementada por las diferentes clases que permiten el acceso a los elementos; *java.lang.reflect.Field, java.lang.reflect.Method*, etc. Es muy fácil de entender si se ha trabajado con la *Reflection API.*

El objeto obtenido que representa la anotación realmente la implementa (la anotación es una interfaz).

1.4 *Autoboxing*

Los tipos primitivos en un lenguaje orientado a objetos como *Java* tienen sentido por eficiencia y comodidad. Sin embargo cada uno tiene su correspondiente clase *wrapper* de uso necesario en algunas ocasiones, por ejemplo para trabajar con colecciones.

La conversión de un valor de tipo primitivo a una instancia de su clase *wrapper* (*boxing*) y al contrario (*unboxing*) es una tarea engorrosa. En *J2SE 5.0* ambas conversiones son automáticas. A esta nueva facilidad se le llama *Autoboxing*.

Ejemplo sencillo:

Hasta ahora:

```
Integer iObject= new Integer(1);      // boxing
int i= iObject.intValue();            // unboxing
```

Con *autoboxing:*

```
Integer iObject= 1;                 // auto-boxing
int i= iObject;                     // auto-unboxing
```

El único pero que se le puede encontrar es que a los nuevos programadores (los que comiencen con *Java5*) esto les pueda confundir y no distingan bien entre valores primitivos y objetos.

El *autoboxing*, aunque es muy sencillo de entender, tiene algunas implicaciones y efectos laterales no tan evidentes, a tener en cuenta:

- Si en un *auto-unboxing* el objeto a convertir es *null*, se producirá una *NullPointerException* ya que el tipo primitivo no puede aceptar ese valor. Ejemplo:

```
Integer iObject= null;
int i= iObject;            // Se lanza una NullPointerException
```

- Las operaciones de los tipos primitivos están aparentemente disponibles para las clases *wrapper*,

aunque realmente lo que pasa es que se aplica el *autoboxing*. Ejemplos:

```
Integer iObject= 0;
Integer iObject1= 1;
Integer iObject2= iObject+iObject1;
iObject2++;
```

```
Boolean boolObject = true;
boolean bool = false;
Boolean result = (boolObject || bool) && boolObject;
```

Aunque esto resulta muy cómodo hay que tener en cuenta que las conversiones tienen un alto coste computacional.

- Hay que tener mucho cuidado con el comparador de igualdad "==". Como se sabe, el tratamiento de este operador para los tipos primitivos y las clases es diferente. En los primitivos compara el valor y en los objetos la dirección en memoria, para distinguir instancias.

  Así que con este operador sólo hay *auto-unboxing* cuando uno de los operandos es primitivo, en caso contrario se comparan objetos. Ejemplos:

```
Integer iObjeto1 = 1000;
Integer iObjeto2 = 1000;

int i1= 1000;
int i2= 1000;


(i2 == i1)
// resultado= true

(i2 == iObjeto1)
// resultado= true, auto-unboxing de iObjeto1

(iObjeto2 == iObjeto1)
// resultado= false, no hay auto-unboxing, comparacion de
objetos
```

Incluso se puede complicar más; los compiladores utilizan para valores que pueden ser representados por un *byte* (-127 a 127 en enteros) objetos *wrapper* inmutables. Así para representar por ejemplo el *Integer(1)* utilizan siempre la misma instancia.

Por tanto el último caso del ejemplo anterior no tendría el mismo resultado. Ejemplo:

```
Integer iObjeto1 = 1;
Integer iObjeto2 = 1;

(iObjeto2 == iObjeto1)
```

```
// resultado= true, no hay auto-unboxing, pero iObjeto2 y
iObjeto1 son la misma instancia
```

Lo mejor en el caso de querer comparar valores (lo normal con tipos primitivos) es forzar la conversión de los objetos a tipos primitivos.

- El *autoboxing* afecta a la resolución de métodos (*method resolution*). Cuando hay sobrecarga de métodos la resolución se basa en los argumentos, pero con la aparición del *autoboxing* se pueden dar casos de ambigüedad. Ejemplo:

```
public static void nombreMetodo(int i)
public static void nombreMetodo(Integer iObjeto)

// LLamada al metodo, no se aplica autoboxing
NombreClase.nombreMetodo(1);
```

En estos casos de ambigüedad, no se aplica el *autoboxing;* deshace la ambigüedad y da compatibilidad hacia atrás.
En caso de no haberla, se aplica el *autoboxing* normalmente. Ejemplo:

```
public static void nombreMetodo(Integer iObjeto)

// LLamada al metodo con autoboxing
NombreClase.nombreMetodo(1);
```

1.5 varargs

varargs se denomina a la posibilidad de declarar que un método admita varios argumentos de un mismo tipo sin determinar la cantidad exacta. Como se puede imaginar es una nueva facilidad del lenguaje *Java*.

La sintaxis es: *NombreTipo**...** nombreArgumento*

Los tres puntos especifican que el argumento es de longitud variable. No tienen que estar justo seguidos del nombre del tipo, pueden estar separados por espacios o tabuladores.

Ejemplo:

```
public void enviarEmail(String asunto,String... emails)
```

El argumento *emails* es de longitud variable, se puede pasar el número de argumentos de tipo *String* que se quiera. Ejemplo de llamada al método:

```
enviarEmail("Saludo","pepe@dominio.com","juan@dominio.com");
```

Restricciones y notas

- Sólo puede haber un argumento variable en un método.
- El argumento variable tiene que ser el último del método.
- Es posible no proporcionar valor alguno para el argumento variable.
  Ejemplo:

```
enviarEmail("Saludo");
```

La llamada anterior es correcta, aunque si se quisiera asegurar una dirección de email, habría que cambiar la definición del método:

```
public void enviarEmail2(String asunto,String email,String...
masEmails)
```

Acceso a los valores

Antes de ver el acceso a los valores del argumento dentro del método, me parece interesante entender cómo está implementado internamente. El compilador determina qué valores de los pasados al método corresponden al argumento variable y forma un *array* con ellos.

De hecho es posible pasar directamente un *array* del tipo correspondiente y el compilador lo acepta sin problemas. Ejemplo:

```
String[] emailsArray= {"pepe@dominio.com","juan@dominio.com"};
enviarEmail("Saludo",emailsArray);
```

Por consiguiente, para acceder a los valores del argumento variable dentro del método lo que hay que hacer es trabajar cómodamente con el *array* de valores. Ejemplo:

```
public void enviarEmail(String asunto,String... emails){

     for (int i = 0; i < emails.length; i++) {
          ..
     }
}
```

1.6 Bucle *for-each*

La nueva sentencia de bucle *for-each* (también llamada *for / in* o *for* mejorado) proporciona una sintaxis muy sencilla para recorrer los elementos de un *array* o cualquier estructura que implemente el interfaz *Iterable* (por ejemplo las colecciones).

Ni que decir tiene que la reducción de código mejora la legibilidad y evita errores.

La simplicidad se logra por la iteración automática sobre los elementos. Esto tiene como contrapartida una pérdida de flexibilidad, pero para eso sigue disponible la sentencia *for* tradicional.

1.6.1 Uso

Forma general de uso:

```
for (TipoElementos  elementoAuxiliar : estructuraElementos) {
        .....
}
```

El elemento auxiliar tiene que ser de un tipo compatible con los elementos de la estructura. En la iteración le van siendo asignados los elementos. En el cuerpo del bucle sólo es necesario utilizar el elemento auxiliar para trabajar con los elementos de la estructura.

También es frecuente acompañar al tipo del elemento auxiliar del modificador *final* para asegurar que los elementos no son modificados en el cuerpo del bucle.

Ejemplos que comparan el bucle clásico *for* con el nuevo *for-each*:

Iteración típica sobre un *array*:

```
for (int i= 0; i < array.length; i++){
     String elemento = array[i];
     ....
}
```

Iteración típica sobre una colección (implementa *Iterable*). En este caso un conjunto:

```
for(Iterator<String> iter= conjunto.iterator();iter.hasNext();)
{
     String elemento= iter.next();
     ....
}
```

Iteración con el nuevo bucle *for*-each. Se unifica la forma para *arrays* y objetos *"Iterables"*:

```
for (String elemento : estructura) {
     ....
}
```

1.6.2 Interfaz *Iterable*

Iterable es una nueva interfaz creada para permitir a las clases (con su implementación) ser soportadas por la sentencia *for-each*. Lo que se consigue realmente es que la clase devuelva un objeto *"Iterator"* que posibilita recorrer los elementos de la clase de una forma establecida.

Por tanto si se quiere crear una estructura propia que sea soportada por *for-each*, tendrá que implementar *Iterable* y como consecuencia devolver un objeto que implemente la interfaz *java.util.Iterator*.

Definición de ambas interfaces:

```
public interface Iterable<E> {
     java.util.Iterator<E> iterator(  );
}
```

```
public interface Iterator<E> {
     boolean hasNext(  );
     E next(  );
     void remove(  );  // Implementacion opcional
}
```

Ejemplo de pila que implementa *Iterable*, devuelve un objeto *java.util.Iterator* con los elementos de la pila ordenados de la cima al fondo:

```
import java.util.*;

public class PilaIterable<E> implements Iterable<E>{

     List<E> pila= new LinkedList<E>();


     void aniadir(E elemento){
          pila.add(elemento);
     }


     E extraer(){
          int i= pila.size()-1;
          E elemento;
          if (i < 0){
               elemento= null;
          }else{
               elemento= pila.get(i);
               pila.remove(i);
          }

          return elemento;
     }
```

```
    boolean estaVacia(){
        return pila.isEmpty();
    }


    public Iterator<E> iterator(){

        return new PilaIterator();
    }


    // Clase interna que implementa la interfaz Iterator

    class PilaIterator implements Iterator<E> {

        int posicion= pila.size();


        public boolean hasNext(){

            return  (posicion != 0);
        }


        public E next(){

            posicion-= 1;
            return pila.get(posicion);
        }


        public void remove(){

            throw new UnsupportedOperationException( );
        }
    }
}
```

1.7 Importación de miembros estáticos (*static imports*)

En *J2SE 5.0* es posible importar miembros estáticos de un tipo, a esta nueva funcionalidad se le denomina *static imports*. Evita anteponer el nombre del tipo a cada aparición del miembro en el fichero.

Es quizás la novedad del lenguaje menos importante, pero en los casos donde se repiten variables o métodos estáticos es conveniente utilizarla ya que descarga el código.

Uso

Como las importaciones de tipos, las de miembros estáticos pueden tener dos formas:

- Importación de un sólo miembro.

  Forma general: *import static identificadorMiembro;*

  Como ejemplo se puede tomar la variable estática *java.lang.System.out*, que permitiría en un

programa abreviar el método de salida por pantalla:

```
import static java.lang.System.out;

....

     out.println(mensaje);
```

- Importación de los miembros de un tipo, usando comodín.

  Forma general: *import static identificadorTipo.*;*

  Ejemplo de hipotética clase que utiliza intensivamente operaciones matemáticas (métodos estáticos) de la clase *java.lang.Math*:

```
import static java.lang.Math.*;

....

     double d= sqrt(cos(abs(23.45)));
```

En este último ejemplo se ve claro cuándo puede ser bastante conveniente su uso.

Otra situación típica de uso de *static imports* es con constantes, que en *Java5* se implementan normalmente con tipos enumerados. Esto permitirá no tener que anteponer el nombre del tipo a los valores.

Ejemplo con el tipo enumerado *NotasMusicales*, definido en aquel capitulo:

```
import static paqueteDeNotasMusicales.NotasMusicales.*;

....

      NotasMusicales nota1= FA;
```

Importación de nombres de miembros.

Hasta ahora por simplificar se ha hablado de importación de miembros estáticos pero realmente se importan nombres de miembros, no miembros concretos. Así que en el caso importar el nombre de un método sobrecargado se están importando todos los métodos que comparten nombre.

Un caso extremo sería si se importarán dos miembros de clases diferentes pero que compartieran nombre y parámetros. En este caso el compilador genera un error alegando ambigüedad del método.

1.8 Otras mejoras menores

1.8.1 Operador condicional

El operador condicional o ternario, forma general:

expresionCondicional ? expresion1 : expresion2

también trae algunas novedades:

- La expresión condicional, debido al *auto-unboxing*, puede ser un objeto *Boolean*.
- El tipo del resultado de la operación es la "intersección" de los tipos de los operandos. Por ejemplo su superclase si heredan de una clase común, o una interfaz que implementan ambos. Esto tiene como consecuencia que con cualquier par de operandos siempre se puede asignar el resultado a un *Object*.

  Ejemplos:

```
String s= "palabra";
StringBuffer sb= new StringBuffer("palabra modificable");

CharSequence cs = true ? sb : s;
```

  En este ejemplo ambas clases implementan la interfaz *CharSequence*

```
Object o= true ? new Integer(39) : "casa";
```

  Aquí los tipos *Integer* y *String* sólo tienen en común que descienden del tipo raíz *Object*.

1.8.2 Sobreescritura del tipo de retorno

Hasta ahora no era posible cambiar el valor de retorno al sobreescribir un método, en *Java 5* es posible con la condición de que el nuevo valor de retorno sea derivado del original.

2 Mejoras en la librería estándar

En esta sección se explican escuetamente las mejoras introducidas en la librería estándar de *J2SE 5.0* .
Para entrar en profundidad, la documentación del *API* es una buena opción.

Utilidades

StringBuilder

Nueva clase que modela una secuencia modificable de caracteres. El objetivo de esta clase es sustituir a la clase *StringBuffer* en la programación normal (no concurrente). *StringBuffer* es una clase sincronizada, característica que la hace ineficiente cuando no se necesita.

Formatter

Se ha añadido un intérprete para el formateo de texto al estilo *printf* del lenguaje *C*. La sintaxis es muy parecida aunque tiene algunas peculiaridades propias de *Java*.

La clase principal es *java.util.*Formatter. Una interfaz muy relacionada es *Appendable*.

Para hacer más sencilla la programación se ha dotado a clases como *java.io.PrintStream, java.io.PrintWriter, String*, de los métodos:

format(String format, Object... args)

format(Locale l, String format, Object... args)

para acceder al formateo directamente, sin tener que crear un objeto *java.util.Formatter*.

java.util.Arrays

Esta clase compuesta por métodos estáticos que facilitan el trabajo con *arrays* se ve ampliada con nuevos métodos. Algunos son:

- *toString()* Devuelve la representación del *array*.
- *deepToString()* Devuelve la representación del *array*, soporta *arrays* multidimensionales.
- *deepEquals()* Comparación de *arrays* multidimensionales.

java.util.Queue

Nueva interfaz genérica que modela la estructura de cola, como colección de elementos en espera a ser procesados. Forma parte del *framework Collections*.

También se incorporan varias clases que implementan la interfaz como *java.util.PriorityQueue*, *java.util.SynchronousQueue*...

java.util.Scanner

Nueva clase que es capaz de escanear un texto y extraer tipos primitivos o *Strings* siguiendo una expresión regular.

Collection framework

Mucho de lo estudiado hasta ahora tiene que ver con las colecciones, lo que quiere decir que ha sido una de las partes de la librería con más cambios:

- Tres de las nuevas características del lenguaje (genéricos, *for-each* y *autoboxing*) están muy relacionadas con las colecciones.
- Tres nuevas interfaces: *java.util.Queue, java.util.concurrent.BlockingQueue* y *java.util.concurrent.ConcurrentMap* y varias implementaciones de estas interfaces.
- Implementaciones de *java.util.Map* y *java.util.Set* específicas para el uso con tipos enumerados.
- Implementaciones de *java.util.List* y *java.util.Set* específicas para *copy-on-write*.
- Implementaciones *wrapper* para proporcionar seguridad de tipos dinámica a la mayoría de los tipos.
- Métodos que implementan nuevos algoritmos para la manipulación de colecciones.
- Métodos para generar códigos *hash* y representaciones de texto de *arrays*.

Internacionalización

La nueva versión de *Java* soporta *Unicode 4.0*, lo que hace insuficientes los *16 bits* que se utilizaban hasta ahora para codificar caracteres. Algunos caracteres, llamados caracteres suplementarios (*suplementary characters*), que no son alcanzados por los 16 *bits* son representados con el tipo *int*.

Esto tiene bastantes implicaciones a nivel interno pero también en lo que respecta a la interfaz de

programación, clases como *Character, String*, el paquete *java.text* se han visto afectadas.

Networking

En cuanto a la programación de redes, hay algunas novedades como la posibilidad de establecer *timeouts*, la mejora en el manejo de c*ookies*, facilidades para comprobar el alcance de un *host*, etc. Ninguna realmente importante.

Seguridad

J2SE 5.0 presenta notables mejoras en lo que a seguridad respecta; soporta más estándares (*SASL, OCSP, TSP*) , mejora el rendimiento y la escalabilidad, mejoras en criptografía y *Java GSS*, etc.

Reflection

La mayoría de las mejoras tienen que ver con el soporte de todas las novedades del lenguaje; es posible saber si un tipo es genérico y todos los detalles al respecto, se puede obtener información sobre los elementos anotados, determinar si un tipo es enumerado, si un método tiene argumento variable, etc. Todo lo necesario relativo a las nuevas características del lenguaje.

También se han añadido métodos para facilitar la programación y se ha hecho genérico el tipo *Class*.

Java API for XML Processing (JAXP)

En *J2SE 5.0* introduce una nueva versión *JAXP 1.3* (en vez de *JAXP 1.1* en *J2SE 1.4*) y una nueva implementación de referencia basada en *Apache Xerces* (en vez de *Crimson* en *J2SE 1.4*).

Estos grandes cambios han traído algunas incompatibilidades con las que hay que tener cuidado.

JAXP 1.3 mejora en varios aspectos la anterior versión, algunos son:

- Proporciona un validador para *XML Schema*.
- Posibilidad de usar otros esquemas de validación (paquete *javax.xml.validation*)
- Soporte de *DOM Level 3*
- Nueva *API* amigable para el uso de expresiones *Xpath* (paquete *javax.xml.xpath*)

Serialización

Se ha añadido soporte a la serialización de tipos enumerados, que difiere de la del resto objetos serializables por sus particularidades.

Programación concurrente

En lo relativo a la programación concurrente, la nueva versión de *Java* trae nuevas e importantes utilidades recogidas en los paquetes *java.util.concurrent*, *java.util.concurrent.atomic* y *java.util.concurrent.locks*.

Estas utilidades generan un entorno para el desarrollo de aplicaciones concurrentes potente, escalable y extensible, gracias a colecciones síncronas, semáforos, variables atómicas, etc..

También proporcionan primitivas de bajo nivel para programación concurrente avanzada llegando a un grado sólo posible hasta ahora con programación nativa.

La clase *Thread* también ha sido mejorada en varios aspectos.

Librerías de integración

Respecto a *JDBC*, la interfaz *javax.sql.RowSet* ha sido implementada de las cinco formas más comúnmente usada.

RMI, JNDI, CORBA, *Java IDL*, y *Java RMI-IIOP* también han sido mejorados.

Swing

Por fin es posible en *Swing* sustituir *jFrame.getContentPane().add()* por *jFrame.add()*.

También se ha añadido soporte de impresión para *JTable*.

Por último se han incorporado dos nuevos *look and feel*: *Synth* y *Ocean*.

Java Management Extensions (JMX) 1.2

En *J2SE 5.0* se incluye el soporte a *JMX* en la librería estándar, hasta ahora era una librería independiente.

3 Maquina virtual

La *Java Virtual Machine* también trae mejoras en la nueva versión:

Class Data Sharing

Con esta nueva tecnología se logra una reducción del tiempo de inicio de las aplicaciones, sobre todo de las pequeñas, y un menor uso de memoria.

Class Data Sharing consiste en la precarga en memoria de un conjunto de clases muy utilizadas, que seran compartidas por todos los procesos. Esta orientado a aplicaciones cliente y por ahora sólo es soportado por *Java HotSpot Client VM* y con el *serial garbage collector*.

La tecnología se habilita automáticamente cuando se dan las condiciones necesarias, sin embargo es posible su control manual.

Garbage Collector

El *parallel collector* ha sido mejorado para adaptarse a las necesidades de memoria de la aplicación. Se puede especificar el tipo de rendimiento que queremos en la aplicación y automáticamente la maquina virtual optimizará el tamaño del *heap* para conseguir los objetivos de rendimiento.

Con esto se evita el ajuste manual por línea de comandos que se necesitaba hasta ahora.

Detección de servidor

Al arrancar una aplicación la máquina virtual intenta adivinar si la máquina en la que se va a ejecutar es un servidor y en ese caso usa *Java HotSpot Server Virtual Machine*.

La deducción se basa en la configuración de la plataforma (*hardware* + sistema operativo), algo normalmente poco fiable y variable en el tiempo.

Otras

- Cambio en el mapeo de la prioridad de los hilos. Ahora los hilos *Java* y los nativos compiten en igualdad.
- El mecanismo de aviso de errores fatales ha mejorado en el diagnóstico y fiabilidad.
- Soporte de granularidad de nanosegundos para mediciones de tiempos (dependiente de plataforma).

4 Mejoras en despliegue (d*eployment*)

Las mejoras son abundantes, especialmente en *Java Web* Start. Veamos:

- Muchas funcionalidades comunes de *Java Web Start* y *Java Plug-in* se han unificado, y comparten panel de control: *Java Control Panel*.
- Mejoras en seguridad de acceso y control más flexible de la seguridad.
- Nuevo formato de compresión de ficheros *jar*, con mayor ratio de compresión, llamado *Pack200*.
- *Java Web Start*:
  - Eliminación del *Developer* Bundle. Ahora los recursos de *JWS* están integrados en el *JRE* y el *JDK*.
  - Integración en el escritorio; se ha extendido al escritorio *GNOME* y se ha mejorado en general.
  - Ahora un fichero *JNLP* puede solicitar cualquier argumento a la *JVM*. Hasta ahora los argumentos que aceptaba eran limitados.
  - Soporte completo a las *Java Printing APIs*.

5 Herramientas y arquitectura creación de herramientas

Java Virtual Machine Tool Interface (JVMTI)

Traducido como interfaz de la *JVM* para herramientas, es una interfaz de programación en el lenguaje nativo de la plataforma (normalmente *C*) destinado a la creación de herramientas de desarrollo (*profiler, debugger*) y monitorización.

Proporciona facilidades para la inspección del estado y control de ejecución de las aplicaciones que corren en la *JVM*.

JVMTI sustituye las ahora obsoletas *JVMPI* y *JVMDI* presentes en la versiones anteriores de *J2SE*.

Java Platform Debugger Architecture (JPDA)

JPDA es una arquitectura para depuración multicapa (la de bajo nivel es *JVMTI*) que facilita la creación de herramientas de depuración independientes de plataforma.

La nueva versión trae muchas mejoras: flexibilidad, soporte a las nuevas características del lenguaje Java, subconjunto de *JDI* de sólo lectura, etc.

Compilador (*javac*)

Nuevas opciones del compilador:

- *-source 1.5*

  Habilita las nuevas características del lenguaje para ser usadas en el código fuente. (*- target 1.5* está implícito)

- *- target 1.5*

  Permite al compilador usar las nuevas funcionalidades de librerías y *JVM*.

- *- Xlint*

  Habilita al compilador para reportar mensajes de aviso sobre legales pero desaconsejadas y potencialmente problemáticas prácticas de programación. Muchas relativas al conflicto entre las nuevas características del lenguaje y la compatibilidad hacia atrás, comentadas en este texto.

- *-d32*

  Indica una plataforma *Linux* o *Solaris* de 32 *bits*

- *-d64*

  Indica una plataforma *Linux* o *Solaris* de 64 *bits*

- *-target cldc1.0*

  Genera *class files* adecuado para el uso en *JVMs* en *Connected Limited Device Configuration (CLDC)* versión 1.0 y superiores.

Herramienta *Javadoc*

La herramienta de documentación también ha sufrido bastantes cambios. Para empezar da soporte para la documentación de los nuevos elementos que se han incorporado al lenguaje (enumerados, anotaciones, etc) , hay nuevas tags (*{@literal} , {@code} , {@value arg}*), la nueva anotación *@Deprecated* ya comentada en el capitulo de anotaciones, etc.

Annotation Processing Tool (apt)

apt es una nueva herramienta de línea de comandos para el proceso de anotaciones. Proporciona un conjunto de *APIs* reflexivas e infraestructura de soporte para el proceso de anotaciones.

Como las anotaciones es algo nuevo en *J2SE*.

JConsole

JConsole es una herramienta gráfica que permite la monitorización y gestión de aplicaciones utilizando la tecnología *JMX*.
La herramienta se distribuye con el *JDK* de *Sun*.

Para terminar, destacar que la mayoría de entornos de desarrollo ya soportan estas mejoras permitiéndonos aprovecharlas con la mayor productividad. Dos de los más extendidos las soportan desde sus versiones *NetBeans 4.0* y *Eclipse 3.1*.

Bibliografía

- David Flanagan, Brett McLaughlin. Java 1.5 Tiger: A Developer's Notebook. O'Reilly 2004
- David Flanagan. Java in a Nutshell, 5th Edition. O'Reilly 2005
- http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html
- http://www.onjava.com/pub/a/onjava/2005/04/20/javaIAN5.html
- http://java.sun.com/developer/technicalArticles/J2SE/5reasons.html
- http://www.onjava.com/pub/a/onjava/2005/07/06/generics.html