Guía práctica de estudio 03:
Utilerías y clases de uso general

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 03:
Utilerías y clases de uso general

Objetivo:

Utilizar bibliotecas propias del lenguaje para realizar algunas tareas comunes y recurrentes.

Actividades:

- Conocer las bibliotecas del lenguaje.
- Utilizar algunas clases propias de la biblioteca del lenguaje.

Introducción

Al trabajar en un problema de programación, normalmente se debe verificar si hay clases pre-construidas que satisfagan las necesidades del programa. Si existen esas clases, entonces hay que utilizarlas: ***"no tratar de reinventar la rueda"***.

Hay dos ventajas principales de usar clases pre-construidas: se puede ahorrar tiempo ya que no es necesario escribir nuevas; y el uso de clases pre-construidas también puede mejorar la calidad de los programas ya que han sido probadas completamente, depuradas y sometidas a un proceso de escrutinio para asegurar su eficiencia.

NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

Arreglos

Un **arreglo** es un objeto en el que se puede almacenar un conjunto de datos de un **mismo tipo**. Cada uno de los elementos del arreglo tiene asignado un **índice** numérico según su posición, siendo 0 el primer índice. Se declara de la siguiente manera:

tipoDeDato [ ] nombreVariable; o *tipoDeDato nombreVariable[ ];*

Como se puede apreciar, los corchetes pueden estar situados delante del nombre de la variable o detrás. Ejemplos:

int [ ] k; *String [ ] p;* *char datos[ ];*

Los arreglos pueden declararse en los mismos lugares que las variables estándar. Para asignar un tamaño al arreglo se utiliza la expresión:

variableArreglo = ***new*** *tipoDeDato[tamaño];*

También se puede asignar tamaño al arreglo en la misma línea de declaración de la variable.

int [ ] k = new int[5];

Cuando un arreglo se dimensiona, todos sus elementos son inicializados explícitamente al **valor por defecto** del tipo correspondiente.

Para declarar, dimensionar e inicializar un arreglo en una misma sentencia se indican los valores del arreglo entre llaves y separados por comas. Ejemplo:

int [ ] nums = {10, 20, 30, 40};

El acceso a los elementos de un arreglo se realiza utilizando la expresión:

variableArreglo[índice]

Donde índice representa la posición a la que se quiere tener acceso y cuyo valor debe estar entre **0** y **tamaño – 1**

Todos los objetos arreglo exponen un atributo publico llamado **length** que permite conocer el tamaño al que ha sido dimensionado un arreglo.

Ejemplo:

```
int [ ] nums = new int[10];
for(int i=0; i < nums.length; i++)
        nums[i] = i * 2;
```

Los arreglos al igual que las variables, se pueden usar como argumentos, así como también pueden ser devueltos por un método o función.

En Java se puede utilizar una variante del for llamado **for-each**, para facilitar el recorrido de arreglos y colecciones recuperando su contenido y eliminando la necesidad de usar una variable de control que sirva de índice. Su sintaxis es:

```
for(tipoDato variable: variableArreglo){
        //instrucciones
}
```

Ejemplo:

```
int [ ] nums = { 4, 6 ,30, 15 };
for(int n : nums){
        System.out.println(n);
}
```

En este caso, sin acceder de forma explícita a las posiciones del arreglo, cada una de estas es copiada automáticamente a la variable auxiliar **n** al principio de cada iteración.

Los arreglos en Java también pueden tener más de una dimensión, al igual que en C/C++ para declarar un arreglo bidimensional ser tendrían que usar dos pares de corchetes y en general para cada dimensión se usa un nuevo par de corchetes.

Ejemplo:

```
int [ ] [ ] matriz;
```

Argumentos por línea de comandos

Es posible suministrar parámetros al método **main** a través de la línea de comandos. Para ello, los valores a pasar deberán especificarse a continuación del nombre de la clase separados por un espacio:

```
>> java NombreClase arg1 arg2 arg3
```

Los datos llegarán al método **main** en forma de un arreglo de cadenas de caracteres.

Ejemplo:

```
public class Ejemplo {

    public static void main(String[] args) {
        System.out.println(args[0]);
        System.out.println(args[1]);
        System.out.println(args[2]);
    }

}
```

Se ejecuta utilizando la siguiente expresión en la línea de comandos:

API de JAVA

La API Java (**Application Programming Interface**) es una interfaz de programación de aplicaciones provista por los creadores del lenguaje, que da a los programadores los medios para desarrollar aplicaciones Java.

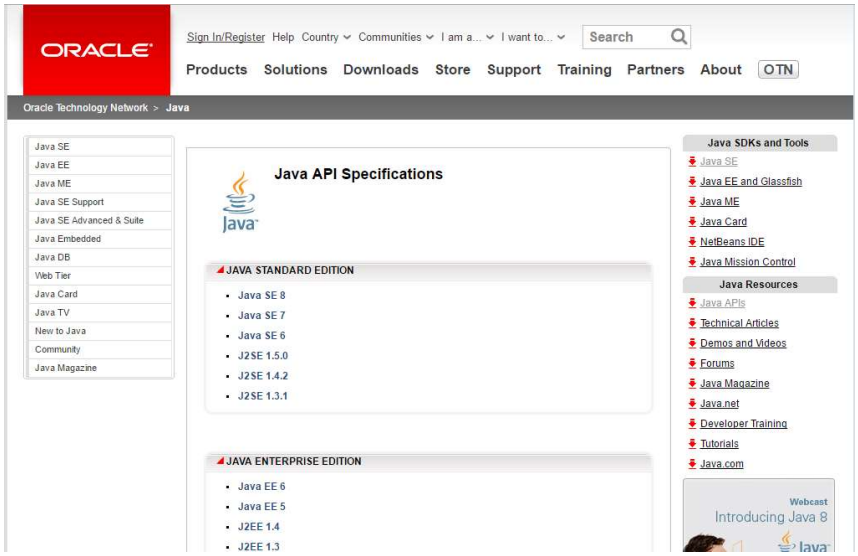
Como el lenguaje Java es un lenguaje orientado a objetos, la **API** de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. La **API** Java está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente.

La información completa del **API** de java se denomina **especificación** y sirve para conocer cualquier aspecto sobre las clases que contiene la **API**. Esta especificación es de crucial importancia para los programadores ya que en ella se pueden consultar los detalles de alguna clase que se quiera utilizar y ya no sería necesario memorizar toda la información relacionada.

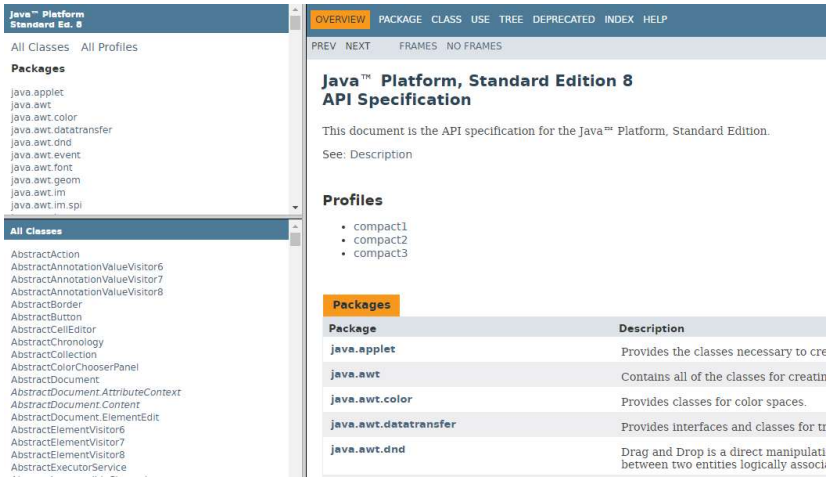
La especificación del **API** de java se encuentra en el sitio de Oracle.

http://www.oracle.com/technetwork/java/api-141528.html

En este sitio se puede consultar la especificación de las últimas versiones (ya que en cada versión se agregan o modifican algunas clases).



Una vez seleccionada la versión, se puede consultar el detalle de todas las clases que integran el API.



Manejo de cadenas

En Java las cadenas de caracteres no son un tipo de datos primitivo, sino que son objetos pertenecientes a la clase **String**.

La clase **String** proporciona una amplia variedad de métodos que permiten realizar las operaciones de manipulación y tratamiento de cadenas de caracteres habituales en un programa.

Para crear un objeto String podemos seguir el procedimiento general de creación de objetos en Java, utilizando el operador new. Ejemplo:

*String s = **new String**("Texto de prueba");*

Sin embargo, dada la amplia utilización de estos objetos en un programa, Java permite crear y asignar un objeto String a una variable de la misma forma que se hace con los tipos de datos primitivos. Entonces el ejemplo anterior es equivalente a:

String s = "Texto de prueba";

Una vez creado el objeto y asignada la referencia al mismo a una variable, puede utilizarse para acceder a los métodos definidos en la clase String (se pueden revisar en la documentación del API com java.lang.String). Los más usados son: length, equals, charAt, substring, indexOf,replace, toUpperCase, toLowerCase, Split, entre otros.

Ejemplo:

s.length(); *//Devuelve el tamaño de la cadena*

s.toUpperCase(); *//Devuelve la cadena en mayúsculas*

Las variables de tipo String se pueden usar en una expresión que use el operador + para concatenar cadenas. Ejemplo:

String s = "Hola";

String t = s + " que tal";

En Java las cadenas de caracteres son objetos **inmutables**, esto significa que una vez que el objeto se ha creado, no puede ser modificado.

Cuando escribimos una instrucción como la del ejemplo anterior, es fácil intuir que la variable **s** pasa a apuntar al objeto de texto *"Hola que tal"*. Aunque a raíz de la operación de concatenación pueda parecer que el objeto *"Hola"* apuntado por la variable **s** ha sido modificado en realidad no sucede esto, sino que al concatenarse *"Hola"* con *" que tal"* se está creando un nuevo objeto de texto *"Hola que tal"* que pasa a ser referenciado por la variable **s**. El objeto *"Hola"* deja de ser referenciado por dicha variable.

Java provee soporte especial para la concatenación de cadenas con las clases **StringBuilder** y **StringBuffer**. Un objeto **StringBuilder** es una secuencia de caracteres **mutable**, su contenido y capacidad puede cambiar en cualquier momento. Además, a diferencia de los **Strings**, los **builders** cuentan con una capacidad (capacity), la cantidad de espacios de caracteres asignados. Ésta es siempre mayor o igual que la longitud (length) y se expande automáticamente para acomodarse a más caracteres.

Los métodos principales de la clase **StringBuilder** son **append** e **insert**. Cada uno convierte un dato en **String** y concatena o inserta los caracteres de dicho String al **StringBuilder**. El método **append** agrega los caracteres al final mientras que **insert** los agrega en un punto específico.

Para hacer la misma concatenación que el ejemplo con String, quedaría:

```
StringBuilder sb = new StringBuilder("Hola");

sb.append(" que tal");
```

Wrappers

Los **wrappers** o clases envoltorio son clases diseñadas para ser un complemento de los tipos primitivos. En efecto, los tipos primitivos son los únicos elementos de Java que no son objetos. Esto tiene algunas ventajas desde el punto de vista de la eficiencia, pero algunos inconvenientes desde el punto de vista de la funcionalidad.

Por ejemplo, los tipos primitivos siempre se pasan como argumento a los métodos por valor, mientras que los objetos se pasan por referencia. No hay forma de modificar en un método un argumento de tipo primitivo y que esa modificación se trasmita al entorno que hizo la llamada.

Una forma de conseguir esto es utilizar un **wrapper**, esto es un objeto cuya variable miembro es el tipo primitivo que se quiere modificar. Las clases **wrapper** también proporcionan métodos para realizar otras tareas con los tipos primitivos, tales como conversión con cadenas de caracteres en uno y otro sentido.

Existe una clase **wrapper** para cada uno de los tipos primitivos: Byte, Short, Character, Integer, Long, Float, Double y Boolean (obsérvese que los nombres empiezan por mayúscula, siguiendo la nomenclatura típica de Java). Todas estas clases se encuentran en ***java.lang.***

Todas las clases **wrapper** permiten crear un objeto de la clase a partir de tipo básico.

```
int k =23;

Integer num = new Integer(k);
```

A excepción de *Character*, las clases **wrapper** también permiten crear objetos partiendo de la representación como cadena del dato.

```
String s = "4.65";

Float ft = new Float(s);
```

Para recuperar el valor a partir del objeto, las ocho clases **wrapper** proporcionan un método con el formato *xxxValue( )* que devuelve el dato encapsulado en el objeto donde xxx representa el nombre del tipo en el que se quiere obtener el dato.

```
float dato = ft.floatValue( );

int n=num.intValue( );
```

Las clases numéricas proporcionan un método estático *parseXxx(String)* que permite convertir la representación en forma de cadena de un numero en el correspondiente tipo numérico donde **Xxx** es el nombre del tipo al que se va a convertir la cadena de caracteres.

```
String s1 = "25", s2="89.2";

int n = Integer.parseInt(s1);

double d = Double.parseDouble(s2);
```

Autoboxing

El **autoboxing** consiste en la encapsulación automática de un dato básico en un objeto wrapper mediante la utilización del operador de asignación.

Por ejemplo:

```
int p = 5;
Integer n = new Integer(p);
```

Equivale a:

```
int p = 5;
Integer n = p;
```

Es decir, la creación del objeto **wrapper** se produce implícitamente al asignar el dato a la variable objeto. De la misma forma, para obtener el dato básico a partir del objeto **wrapper** no será necesario recurrir al método *xxxValue( )*, esto se realizará implícitamente al utilizar la variable objeto en una expresión. A esto se le conoce como **autounboxing**. Para el ejemplo anterior:

```
int a = n;
```

Colecciones

Una **colección** es un objeto que almacena un conjunto de **referencias a objetos**, es decir, es parecido a un arreglo de objetos. Sin embargo, a diferencia de los arreglos, las colecciones son **dinámicas**, en el sentido de que no tienen un tamaño fijo y permiten añadir y eliminar objetos en tiempo de ejecución.

Java incluye un amplio conjunto de clases para la creación y tratamiento de colecciones. Todas ellas proporcionan una serie de métodos para realizar las operaciones básicas sobre una colección, como son:

- Añadir objetos a la colección.
- Eliminar objetos de la colección.
- Obtener un objeto de la colección
- Localizar un objeto en la colección.
- Iterar a través de una colección.

Los principales tipos de colecciones que se encuentran por defecto en el API Java son:

Conjuntos

Un conjunto (**Set**) es una colección desordenada (no mantiene un orden de inserción) y no permite elementos duplicados.

Clases de este tipo: HashSet, TreeSet, LinkedHashSet.

Listas

Una lista (**List**) es una colección ordenada (debido a que mantiene el orden de inserción) pero permite elementos duplicados.

Clases de este tipo: ArrayList y LinkedList

Mapas

Un mapa (**Map** también llamado arreglo asociativo) es un conjunto de elementos agrupados con una llave y un valor:

<llave, valor>

Donde las llaves no pueden ser repetidas y a cada valor le corresponde una llave. La columna de valores sí puede repetir elementos.

Clases de este tipo: HashMap, HashTable, TreeMap, LinkedHashMap.

Algunas de las clases más usadas para manejo de colecciones son las siguientes (todas ellas se encuentran en **java.util**):

ArrayList

Se basa en un arreglo redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones. Para crear un objeto ArrayList se utiliza la siguiente sintáxis:

ArrayList<*TipoDato*> *nombreVariable* = *new* ***ArrayList***<*TipoDato*>*( );*

Ejemplo:

ArrayList<Integer> arreglo = new ArrayList<Integer>( );

En este caso se creó un **ArrayList** llamado **arreglo**, el cual podrá contener elementos enteros (Integer).

Una vez creado, se pueden usar los métodos de la clase ArrayList para realizar las operaciones habituales con una colección, las más usuales son:

- **add(elemento)** – Añade un nuevo elemento al ArrayList y lo sitúa al final del mismo.
- **add(índice, elemento)** – Añade un nuevo elemento al ArrayList en la posición especificada por índice, desplazando hacia delante el resto de los elementos de la colección.
- **get(índice)** – Devuelve el elemento en la posición índice.
- **remove(índice)** – Elimina el elemento del ArrayList recorriendo los elementos de las posiciones siguientes. Devuelve el elemento eliminado.
- **clear( )** – Elimina todos los elementos del ArrayList.
- **indexOf(elemento)** – Localiza en el ArrayList el elemento indicado devolviendo su posición o índice. En caso de que el elemento no se encuentre devuelve -1.
- **size( )** – Devuelve el número de elementos almacenados en el ArrayList.

Ejemplo:

```
import java.util.ArrayList;

public class Colecciones {

    public static void main(String[] args) {
        ArrayList<Integer> arreglo = new ArrayList<Integer>();
        arreglo.add(1);
        arreglo.add(8);
        arreglo.add(5);
        arreglo.add(1, 9);
        System.out.println("Tamaño del array list " + arreglo.size());
        System.out.println("Elemento en la posicion 3: " + arreglo.get(3));
        for (Integer elemento : arreglo) {
            System.out.println(elemento);
        }

    }

}
```

Hashtable

La clase Hashtable representa un tipo de colección basada en claves, donde los elementos almacenados en la misma (valores) no tienen asociado un índice numérico basado en su posición, sino una clave que lo identifica de forma única dentro de la colección. Una clave puede ser cualquier tipo de objeto.

La utilización de colecciones basadas en claves resulta útil en aquellas aplicaciones en las que se requiera realizar búsquedas de objetos a partir de un dato que lo identifica. La creación de un objeto Hashtable se realiza de la siguiente manera:

Hashtable<*TipoDatoClave, TipoDatoElemento*> *nombreVariable = new* ***Hashtable***<*TipoDatoClave, TipoDatoElemento*>*( );*

Ejemplo:

Hashtable<String, Integer> miHashTable = new Hashtable<String, Integer>( );

Los principales métodos de la clase Hashtable para manipular la colección son los siguientes:

- **put(clave, valor)** – Añade a la colección el elemento **valor**, asignándole la **clave** especificada. En caso de que exista esa **clave** en la colección el elemento se sustituye por el nuevo **valor**.
- **containsKey(clave)** – Indica si la **clave** especificada existe o no en la colección. Devuelve un **boolean**.
- **get(clave)** – Devuelve el **valor** que tiene asociado la **clave** que se indica. En caso de que no exista ningún elemento con esa **clave** asociada devuelve **null**.
- **remove(clave)** – Elimina de la colección el **valor** cuya **clave** se especifica. En caso de que no exista ningún elemento con esa **clave** no hará nada y devolverá **null**, si existe eliminará el elemento y devolverá una referencia al mismo.
- **size( )** – Devuelve el número de elementos almacenados en el Hashtable.

Ejemplo:

```
import java.util.Hashtable;

public class Colecciones {

    public static void main(String[] args) {

        Hashtable<String, Integer> miTabla = new Hashtable<String, Integer>();
        miTabla.put("uno", 1);
        miTabla.put("dos",2);
        miTabla.put("cinco", 5);

        System.out.println("Contiene a cuatro? " + miTabla.containsKey("cuatro"));

        for (String clave : miTabla.keySet()) {
            System.out.println(clave);
        }
        for (Integer valor : miTabla.values()) {
            System.out.println(valor);
        }

    }

}
```

Al no estar basado en índices, un Hashtable no se puede recorrer totalmente usando el for con una sola variable. Esto no significa que no se pueda iterar sobre un Hashtable, se puede hacer a través de una enumeración.

Los métodos proporcionados por la enumeración (**Enumeration)** permiten recorrer una colección de objetos asociada y acceder a cada uno de sus elementos.

Así, para recorrer completamente el Hashtable, se obtienen sus claves usando el método *keys( )* y para cada una de las claves se obtiene su valor asociado usando *get(clave)*.

Ejemplo:

```
import java.util.Enumeration;
import java.util.Hashtable;

public class Colecciones {

    public static void main(String[] args) {

        Hashtable<String, Integer> miTabla = new Hashtable<String, Integer>();
        miTabla.put("uno", 1);
        miTabla.put("dos",2);
        miTabla.put("cinco", 5);

        String clave;
        Integer valor;
        Enumeration<String> claves = miTabla.keys();
        while(claves.hasMoreElements()){
            clave = claves.nextElement();
            valor = miTabla.get(clave);
            System.out.println("Clave : " + clave + "\tValor : " + valor);
        }

    }

}
```

Clases de utilerías

En Java existen algunas clases que sirven para apoyar el desarrollo de aplicaciones, dichas clases tienen funcionalidades generales como por ejemplo cálculos matemáticos, fechas, etc. Algunas de las clases más útiles son:

Math

Esta clase proporciona métodos para la realización de las **operaciones matemáticas** más habituales. Para utilizar sus métodos simplemente se utiliza el nombre de la clase Math seguida del operador punto y el nombre del método a utilizar.

Ejemplo:

***Math**.pow(5, 2);* *//Eleva 5 a la potencia 2*

***Math**.sqrt(25);* *//Obtiene la raíz cuadrada de 25*

Date y Calendar

En java.util se encuentran dos clases para el tratamiento básico de fechas: **Date** y **Calendar**.

Un objeto **Date** representa una fecha y hora concretas con precisión de un milisegundo esta clase permite manipular una fecha y obtener información de la misma de una manera sencilla, sin embargo, a partir de la versión 1.1 se incorporó una nueva clase llamada **Calendar** que amplía las posibilidades a la hora de trabajar con fechas.

Para crear un objeto de la clase **Date** con la fecha y hora actual se utiliza:

***Date** fecha = new **Date**( );*

Usando el método *toString( )* se obtiene la representación en forma de cadena de la fecha:

System.out.println(fecha.toString());

Calendar es una clase que surgió para cubrir las carencias de la clase **Date** en el tratamiento de las fechas. Para crear un objeto de **Calendar** se usa la siguiente sintaxis:

***Calendar** calendario = **Calendar**.getInstance( );*

Utilizando el método *get( )* se puede recuperar cada uno de los campos que componen la fecha, para ello este método acepta un numero entero indicando el campo que se quiere obtener. La propia clase **Calendar** define una serie de **constantes** con los valores que corresponden a cada uno de los campos que componen una fecha y hora.

Ejemplo:

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class Fechas {

    public static void main(String[] args) {
        Date fecha = new Date();
        System.out.println(fecha.toString());
        SimpleDateFormat formateador = new SimpleDateFormat("dd-MM-yyyy");
        System.out.println(formateador.format(fecha));

        Calendar calendario = Calendar.getInstance();
        String miFecha = "Hoy es día ";
        miFecha += calendario.get(Calendar.DAY_OF_MONTH) + " del mes ";
        miFecha += calendario.get(Calendar.MONTH)+ 1 + " de ";
        miFecha += calendario.get(Calendar.YEAR);
        System.out.println(miFecha);

    }

}
```

A partir de la introducción de la versión **Java 8**, el manejo de las fechas y el tiempo ha cambiado en Java. Desde esta versión, se ha creado una nueva API para el manejo de fechas y tiempo en el paquete **java.time**, que resuelve distintos problemas que se presentaban con el manejo de fechas y tiempo en versiones anteriores.

Ejemplo:

```
LocalDate hoy = LocalDate.now();
System.out.println(hoy);
System.out.println(hoy.plusWeeks(1));
```

Bibliografía

Martín, Antonio
Programador Certificado Java 2.
Segunda Edición.
México
Alfaomega Grupo Editor, 2008

Sierra Katy, Bates Bert
SCJP Sun Certified Programmer for Java 6 Study Guide
Mc Graw Hill

Dean John, Dean Raymond.
Introducción a la programación con Java
Primera Edición.
México
Mc Graw Hill, 2009