Guía práctica de estudio 06: Organización de clases

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

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 06:

Organización de clases

Objetivo:

Organizar adecuadamente las clases según su funcionalidad o propósito bajo un *namespace* o paquete.

Introducción

Las clases de las bibliotecas estándar del lenguaje están organizadas en jerarquías de paquetes. Esta organización en jerarquías ayuda a que las personas encuentren clases particulares que requieren utilizar.

Está bien que varias clases tengan el mismo nombre si están en paquetes distintos. Así, encapsular grupos pequeños de clases en paquetes individuales permite reusar el nombre de una clase dada en diversos contextos.

Nota:

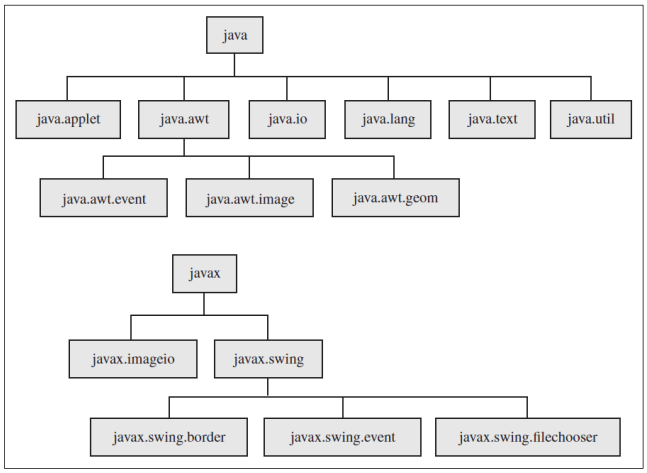
En esta guía se tomará como caso de estudio el lenguaje de programación JAVA. Sin embargo queda a criterio del profesor el uso de este u otro lenguaje orientado a objetos.

Paquetes

Un **paquete** o **package** es una agrupación de clases. La API de Java cuenta con muchos **paquetes**, que contienen clases agrupadas bajo un mismo propósito. Dado que la biblioteca de Java contiene miles de clases, es necesaria alguna estructura en la organización de la biblioteca para facilitar el trabajo con este enorme número de clases.

Java utiliza **paquetes** para acomodar las clases de la biblioteca en grupos que permanecen juntos. Las clases del API están organizadas en **jerarquías de paquetes**. Esta organización en jerarquías ayuda a encontrar clases particulares.

A continuación se muestra parte de la **jerarquía de paquetes** API de Java.



Los **paquetes** se utilizan con las finalidades siguientes:

- Para agrupar clases relacionadas.
- Para evitar conflictos de nombres. En caso de conflicto de nombres entre clases el compilador obliga diferenciarlos usando su nombre cualificado.
- Para ayudar en el control de la accesibilidad de clases y miembros.

El nombre completo o nombre calificado (**Fully Qualified Name**) de una clase debe ser único y está formado por el nombre de la clase precedido por los nombres de los sub-paquetes en donde se encuentra hasta llegar al paquete principal, separados por puntos.

Ejemplo:

java.util.Random es el **Fully Qualified Name** de la clase *Random* que se encuentra en el paquete *java.util*

Importar paquetes

Las clases de Java se almacenan en la biblioteca de clases pero no están disponibles automáticamente para su uso, tal como las otras clases del proyecto actual. Para poder disponer de alguna de estas clases, se debe indicar en el código que se va usar una clase de la biblioteca usando su **fully qualfied name**.

Ejemplo:

```
public class PruebaPaquetes {

    public static void main(String[] args) {
        java.util.Random rnd = new java.util.Random();
        System.out.println(rnd.nextInt(100));
    }

}
```

Para hacer que las clases en un paquete particular estén disponibles para el programa que se está escribiendo, es necesario **importar** ese paquete, esto permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre completo de la clase.

La sentencia **import** tiene la forma general:

import *fullyQualifiedName;*

Estas sentencias deben ir antes de la declaración de la clase. Ejemplo:

```
import java.util.Random;

public class PruebaPaquetes {

    public static void main(String[] args) {
        Random rnd = new Random();
        System.out.println(rnd.nextInt(100));
    }

}
```

Java también permite importar paquetes completos con sentencias de la forma

import *nombreDePaquete.*;*

Por ejemplo, la siguiente sentencia importaría todas las clases del paquete *java.util*:

import *java.util.*;*

El **importar** un paquete no hace que se carguen todas las clases del paquete, sino que sólo se cargarán las clases **public** del paquete.

Al **importar** un paquete **no se importan los sub-paquetes**. Éstos deben ser importados explícitamente, pues en realidad son paquetes distintos.

Por ejemplo, al importar *java.awt* no se importa *java.awt.event*.

Algunas clases se usan tan frecuentemente que casi todas las clases debieran importarlas. Estas clases se han ubicado en el paquete **java.lang** y este paquete se **importa automáticamente** dentro de cada clase. La clase *String* es un ejemplo de una clase ubicada en *java.lang*.

Paquetes propios

El lenguaje Java permite crear sus **propios paquetes** para organizar clases definidas por el programador en jerarquías de paquetes.

Para que una clase pase a formar parte de un **paquete** hay que introducir en ella la sentencia:

package *nombreDelPaquete;*

La cual debe ser la **primera sentencia del archivo** sin contar comentarios y líneas en blanco.

Los nombres de los **paquetes** se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un paquete puede constar de varios nombres unidos por puntos.

Todas las clases que forman parte de un **paquete** deben estar en el mismo directorio. Los nombres compuestos de los paquetes están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los nombres de las clases sean únicos en Internet. Es el nombre del paquete lo que permite obtener esta característica. Una forma de conseguirlo es incluir el nombre del dominio.

Por ejemplo, la clase:

mx.unam.fi.poo.MiClase.class

Debería estar en:

CLASSPATH\mx\unam\fi\poo\MiClase.class

Ejemplo:

```
package mx.unam.fi.poo;

public class MiClase {
    public static void main(String[] args) {
        System.out.println("Clase empaquetada");
    }

}
```

Compilación y ejecución de clases en paquetes

Se puede solicitar al compilador de Java que coloque en forma automática el archivo compilado .class en la ruta destino correspondiente. Para hacer lo anterior, debe invocar al compilador desde línea de comandos con la opción –d, como sigue:

javac -d *rutaOrigen archivoFuente*

El nombre de ruta completo del directorio que obtiene el código compilado es *rutaOrigen/rutaDelPaquete.*

Si el directorio destino ya existe, entonces el archivo generado **.class** va a ese directorio. Si el directorio destino no existe, el compilador crea en forma automática el directorio requerido y luego inserta ahí el archivo generado **.class**. Por tanto, no es necesario crear explícitamente la estructura de directorios, se puede dejar que el compilador lo haga. (IDE típicas también proporcionan formas para hacer lo anterior).

Ejemplo:

Se desea poner la clase *HolaMundo* en un paquete llamado *hola*, para tal efecto, se modifica el código fuente de la siguiente manera:

```
package hola;
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("Hola Mundo");
    }
}
```

Para que se genere el archivo **HolaMundo.class** dentro del directorio hola, se compila con la opción –d y la ruta origen sería el directorio actual, es decir punto (.):

```
D:\>dir hola*
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\

21/06/2016  09:43 p. m.               130 HolaMundo.java
               1 archivos            130 bytes
               0 dirs  741,121,339,392 bytes libres

D:\>javac -d . HolaMundo.java

D:\>dir hola*
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\

21/06/2016  09:47 p. m.    <DIR>          hola
21/06/2016  09:43 p. m.               130 HolaMundo.java
               1 archivos            130 bytes
               1 dirs  741,121,339,392 bytes libres
```

En este caso se generó un directorio llamado hola, dentro del cual se generó el archivo **HolaMundo.class** correspondiente a la clase compilada.

```
D:\>dir hola
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\hola

21/06/2016  09:47 p. m.    <DIR>          .
21/06/2016  09:47 p. m.    <DIR>          ..
21/06/2016  09:47 p. m.               427 HolaMundo.class
               1 archivos            427 bytes
               2 dirs  741,121,339,392 bytes libres

D:\>
```

Para poder ejecutar correctamente esta nueva clase compilada, se debe hacer usando su **Fully Qualified Name** desde la ruta origen, ya que si solo se invoca el intérprete **java** con el nombre de la clase la ejecución fallará dado que no reconoce la clase.

```
D:\>java HolaMundo
Error: no se ha encontrado o cargado la clase principal HolaMundo

D:\>java hola.HolaMundo
Hola Mundo

D:\>
```

Distribución de aplicaciones

Una aplicación en Java está compuesta por varios archivos **.java**. Al compilarlos obtenemos varios archivos **.class** (uno por archivo **.java**), y no un único archivo ejecutable como ocurre en otros lenguajes. Además, a menudo la aplicación está formada no sólo por los archivos **.class** sino que requiere archivos adicionales (como archivos de texto, de configuración, iconos, etc.) lo que multiplica la cantidad de archivos que forman la aplicación compilada.

Todo esto hace que "*llevarse*" la aplicación para ejecutarla en una computadora diferente resulte un poco tedioso, sin mencionar que, olvidar cualquiera de los archivos que componen la aplicación significaría que ésta no va a funcionar correctamente.

Los archivos **JAR** (**Java ARchives**) permiten incluir en un sólo archivo varios archivos diferentes, almacenándolos en un formato **comprimido** para que ocupen menos espacio. Las siglas están deliberadamente escogidas para que coincidan con la palabra inglesa **"jar"** (tarro).

Es por tanto, algo similar a un archivo **.zip** pero con la particularidad de los archivos **.jar no necesitan ser descomprimidos para ser usados**, es decir que el intérprete de Java es capaz de ejecutar los archivos comprimidos en un archivo **JAR** directamente.

Los archivos **JAR**, construidos sobre el formato de archivo **ZIP**, pueden recuperarse o desarrollarse desde cero utilizando los comandos y herramientas **JAR** proporcionadas por el **JDK**.

Una archivo **JAR** incluye una estructura de directorios con clases, lo anterior permite:

- Distribuir/utilizar clases de una manera eficiente a través de un solo archivo.
- Declarar dichas clases de una manera más eficiente en la variable CLASSPATH.
- En todo JDK se incluye el comando jar el cual permite generar, observar y descomprimir archivos JAR;

Los archivos **JAR** contienen archivos de clases y recursos de la aplicación. En general un archivo **JAR** puede contener:

- Los archivos ***.class** que se generan a partir de compilar los archivos ***.java** que componen la aplicación.
- Los archivos de **recursos** que necesita la aplicación (Por ejemplo archivos de configuración, de texto, de sonido, imágenes, etc.)
- Opcionalmente se puede incluir los archivos de código fuente ***.java**
- Opcionalmente puede existir un archivo de configuración "**META-INF/MANIFEST.MF**".

Para que el archivo **JAR** sea ejecutable se debe incluir en el archivo MANIFEST.MF una línea indicando la clase que contiene el método estático *main( )* que se usará para iniciar la aplicación. Este archivo se puede generar manualmente con cualquier editor de texto y es importante destacar que al final de la línea hay que agregar un salto de línea para que funcione. Si el archivo no se crea manualmente, se puede crear con la herramienta **JAR**.

Ejemplo:

Se tiene la clase *MiClase.java*, la cual se encuentra dentro del paquete *mx.unam.fi.poo*:

```
package mx.unam.fi.poo;

public class MiClase {
    public static void main(String[] args) {
        System.out.println("Clase empaquetada");
    }

}
```

Para compilarla y ejecutarla correctamente se utiliza lo siguiente:

```
D:\pruebasJava>dir
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\pruebasJava

15/07/2016  06:26 p. m.    <DIR>          .
15/07/2016  06:26 p. m.    <DIR>          ..
15/07/2016  06:26 p. m.               146 MiClase.java
               1 archivos            146 bytes
               2 dirs  710,417,502,208 bytes libres

D:\pruebasJava>javac -d . MiClase.java

D:\pruebasJava>dir
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\pruebasJava

15/07/2016  06:26 p. m.    <DIR>          .
15/07/2016  06:26 p. m.    <DIR>          ..
15/07/2016  06:26 p. m.               146 MiClase.java
15/07/2016  06:26 p. m.    <DIR>          mx
               1 archivos            146 bytes
               3 dirs  710,417,502,208 bytes libres

D:\pruebasJava>dir mx\unam\fi\poo
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\pruebasJava\mx\unam\fi\poo

15/07/2016  06:26 p. m.    <DIR>          .
15/07/2016  06:26 p. m.    <DIR>          ..
15/07/2016  06:26 p. m.               440 MiClase.class
               1 archivos            440 bytes
               2 dirs  710,417,502,208 bytes libres

D:\pruebasJava>java mx.unam.fi.poo.MiClase
Clase empaquetada
```

Sin embargo, se desea incluir esta clase en un archivo **JAR.** Para tal efecto, se utiliza la herramienta **jar** (incluida con el **JDK** al igual que **javac**).

Además se desea que la herramienta genere el archivo MANIFEST.MF indicando que la clase principal (que contiene el *main*) es **mx.unam.fi.poo.MiClase**, entonces, se utiliza el siguiente comando:

```
D:\pruebasJava>jar -cvfe MiJar.jar mx.unam.fi.poo.MiClase mx/unam/fi/poo/*.class
manifiesto agregado
agregando: mx/unam/fi/poo/MiClase.class(entrada = 440) (salida = 303)(desinflado 31
%)

D:\pruebasJava>dir
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\pruebasJava

15/07/2016  06:32 p. m.    <DIR>          .
15/07/2016  06:32 p. m.    <DIR>          ..
15/07/2016  06:26 p. m.               146 MiClase.java
15/07/2016  06:32 p. m.               824 MiJar.jar
15/07/2016  06:26 p. m.    <DIR>          mx
               2 archivos            970 bytes
               3 dirs  710,417,498,112 bytes libres
```

Con esto se genera el archivo **MiJar.jar**, dentro del cual se encuentra la jerarquía de directorios correspondientes al paquete ***mx.unam.fi.poo*** con el archivo compilado ***MiClase.class*** y adicionalmente el archivo **MANIFEST.MF** dentro de un directorio **META-INF** (ambos creados por la herramienta) el cual contiene lo siguiente;

Manifest-Version: 1.0
Created-By: 1.8.0_45 (Oracle Corporation)
Main-Class: mx.unam.fi.poo.MiClase

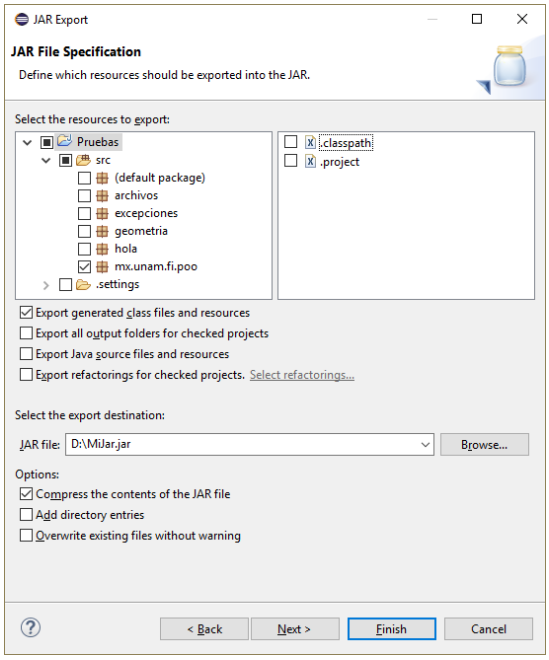
Para verificar el contenido del archivo **JAR** se puede usar el siguiente comando:

```
D:\pruebasJava>jar tf MiJar.jar
META-INF/
META-INF/MANIFEST.MF
mx/unam/fi/poo/MiClase.class
```

Y finalmente para ejecutar el método *main* de la clase principal (en este caso *MiClase*) se utiliza el comando **java –jar** seguido del nombre del archivo JAR:

```
D:\pruebasJava>java -jar MiJar.jar
Clase empaquetada
```

Si se está utilizando algún **IDE**, la generación de los archivos **JAR** es más sencilla ya que generalmente se cuenta con un proceso guiado paso a paso (**wizard**).



La utilización de archivos **JAR** no solo facilita la distribución de clases, sino también su uso por parte de otras aplicaciones, ya que no es necesario descomprimir el archivo para que las clases contenidas en el puedan ser utilizadas por otras clases.

Una forma de lograr esto puede ser agregar el archivo **JAR** al CLASPATH del sistema, es decir, la ruta completa donde se encuentra el archivo y el nombre del mismo (por ejemplo, *D:\pruebasJava\MiJar.jar*) con lo cual ya se podrán utilizar las clases contenidas en el **JAR** importándolas con su **Fully Qualified Name**.

Otra opción es utilizar el **claspath en línea**, es decir, a la hora de compilar o ejecutar se indica con la opción **java –classpath** el jar que debe incluir.

Por ejemplo:

javac –classpath *MiJar.jar OtraClase.java*

java –classpath *MiJar.jar OtraClase*

Documentación

La escritura de buena **documentación** de las definiciones de las clases y de las interfaces es un complemento importante para obtener código de buena calidad. La **documentación** le permite al programador comunicar sus intenciones a los lectores humanos en un lenguaje natural de alto nivel, en lugar de forzarlos a leer código de nivel relativamente bajo.

La **documentación** de los elementos **públicos** de una clase o de una interfaz tiene un valor especial, pues los programadores pueden usarla sin tener que conocer los detalles de su implementación.

Java cuenta con una herramienta de documentación llamada **javadoc** que se distribuye como parte del kit de desarrollo (**JDK**). Esta herramienta automatiza la **generación de documentación** de clases en formato **HTML** con un estilo consistente. El **API** de Java ha sido documentado usando esta misma herramienta y se aprecia su valor cuando se usa la biblioteca de clases.

Los comentarios **Javadoc**, están delimitados por **/**** y ***/**. Al igual que con los comentarios tradicionales, el compilador ignora todo el texto entre los delimitadores de los comentarios **Javadoc**.

Los elementos de una clase que se documentarán son:

- La definición de la clase
- Sus campos
- Sus constructores
- Sus métodos

Desde el punto de vista de un usuario, lo más importante de una clase es que tenga **documentación** sobre ella y sobre sus constructores y métodos públicos. Tendemos a no proporcionar comentarios del estilo de **javadoc** para los campos aunque recordamos que forman parte del detalle del nivel de implementación y no es algo que verán los usuarios.

Un **comentario** contendrá una descripción principal seguida por una sección de etiqueta, aunque ambas partes son opcionales.

La **descripción principal** de una clase debiera consistir en una descripción del objetivo general de la clase. La descripción principal de un método debiera ser bastante general, sin introducir demasiados detalles sobre su implementación. En realidad, la descripción principal de un método generalmente consiste en una sola oración.

Ejemplo:

```
/**
        Crea un nuevo pasajero con distintas ubicaciones de
        salida y de destino.
*/
```

Las ideas esenciales debieran presentarse en la primera sentencia de la descripción principal de una clase, de una interfaz o de un método ya que es lo que se usa a modo de resumen independiente en la parte superior de la documentación generada.

Javadoc también soporta el uso de etiquetas HTML en sus comentarios.

A continuación de la descripción principal aparece la sección de **etiquetas**. (**Javadoc** reconoce alrededor de 20 etiquetas). Las etiquetas pueden usarse de dos maneras: en bloques de etiquetas o como etiquetas de una sola línea. Los bloques de etiquetas son los que se usan con mayor frecuencia. Para ver más detalles sobre las etiquetas de una sola línea y sobre las restantes etiquetas, puede recurrir a la sección **javadoc** de la documentación Tools and Utilities que forma parte del JDK.

(http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html)

Las etiquetas más usadas son:

Etiqueta	Texto asociado
`@author`	nombre(s) del autor(es)
`@param`	nombre de parámetro y descripción
`@return`	descripción del valor de retorno
`@see`	referencia cruzada
`@throws`	tipo de excepción que se lanza y las circunstancias en las que se hace
`@version`	descripción de la versión

Las etiquetas *@author* y *@version* se encuentran regularmente en los comentarios de una clase y de una interfaz y no pueden usarse en los comentarios de métodos, constructores o campos. Ambas etiquetas pueden estar seguidas de cualquier texto y no se requiere ningún formato especial para ninguna de ellas. Ejemplos:

@author Hakcer T. LargeBrain
@version 2004.12.31

Las etiquetas *@param* y *@throws* se usan en métodos y en constructores, mientras que *@return* se usa sólo en métodos. Algunos ejemplos:

@param limite El valor máximo permitido.
@return Un número aleatorio en el rango 1 a limite (inclusive)
@throws IllegalLimitException Si el límite es menor que 1.

La etiqueta *@see* adopta varias formas diferentes y puede usarse en cualquier comentario de documentación. Proporciona un camino de referencia cruzada hacia un comentario de otra clase, método o cualquier otra forma de documentación. Se agrega una sección *See Also* al elemento que está siendo comentado. Algunos ejemplos típicos:

@see "The Java Language Specification, by Joy et al"
*@see The BlueJ web site *
@see #estaVivo
@see java.util.ArrayList#add

La primera simplemente encierra un texto en forma de cadena sin un hipervínculo, la segunda es un hipervínculo hacia el documento especificado, la tercera es un vínculo a la documentación del método *estaVivo* de la misma clase, la cuarta vincula la documentación del método *add* con la clase *java.util.ArrayList*.

Para ejecutar **javadoc**, debe introducirse este comando

javadoc -d outputDirectory sourceFiles

La opción ***–d outputDirectory*** hace que la salida vaya a otro directorio. Si se omite la opción ***–d***, por defecto la salida se dirige al directorio actual, pero no es una buena idea, ya que **javadoc** crea muchos archivos que pueden enredar el directorio actual. Es posible colocar documentación para más de una clase en el mismo directorio. Se usan espacios para separar múltiples nombres de archivos fuente con espacios.

Ejemplo:

Para generar la documentación de la clase MiClase, se debe editar su código fuente y agregarle los comentarios correspondientes. Una vez hecho esto se procede a generar la documentación con javadoc.

```java
package mx.unam.fi.poo;

/**
 * Esta es una clase de prueba
 * @author Angelica Nakayama
 * @version Julio-2016
 *
 */
public class MiClase {

    /**
     * Metodo principal o main.
     * Imprime un mensaje indicando que la clase fue empaquetada
     * @param args Los argumentos de linea de comandos
     */
    public static void main(String[] args) {
        System.out.println("Clase empaquetada");
    }

}
```

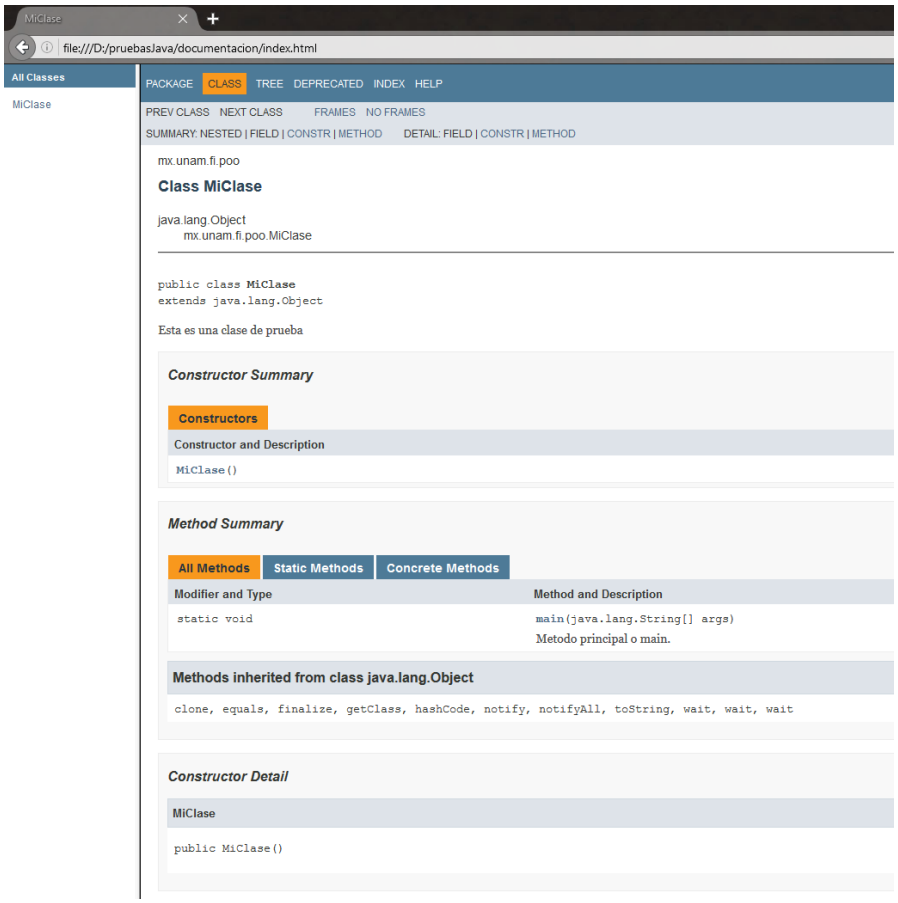
```
D:\pruebasJava>javadoc -d documentacion MiClase.java
Loading source file MiClase.java...
Constructing Javadoc information...
Creating destination directory: "documentacion\"
Standard Doclet version 1.8.0_45
Building tree for all the packages and classes...
Generating documentacion\mx\unam\fi\poo\MiClase.html...
Generating documentacion\mx\unam\fi\poo\package-frame.html...
Generating documentacion\mx\unam\fi\poo\package-summary.html...
Generating documentacion\mx\unam\fi\poo\package-tree.html...
Generating documentacion\constant-values.html...
Building index for all the packages and classes...
Generating documentacion\overview-tree.html...
Generating documentacion\index-all.html...
Generating documentacion\deprecated-list.html...
Building index for all classes...
Generating documentacion\allclasses-frame.html...
Generating documentacion\allclasses-noframe.html...
Generating documentacion\index.html...
Generating documentacion\help-doc.html...

D:\pruebasJava>dir
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\pruebasJava

20/07/2016  11:23 a. m.    <DIR>          .
20/07/2016  11:23 a. m.    <DIR>          ..
20/07/2016  11:22 a. m.    <DIR>          documentacion
15/07/2016  06:32 p. m.               104 MANIFEST.MF
20/07/2016  11:21 a. m.               409 MiClase.java
15/07/2016  06:32 p. m.               824 MiJar.jar
15/07/2016  06:26 p. m.    <DIR>          mx
               3 archivos          1,337 bytes
               4 dirs  710,416,855,040 bytes libres
```

La documentación generada estará en el directorio documentacion y se puede revisar abriendo el archivo index.html con cualquier navegador.



Bibliografía

Barnes David, Kölling Michael
Programación Orientada a Objetos con Java.
Tercera Edición.
Madrid
Pearson Educación, 2007

Deitel Paul, Deitel Harvey.
Como programar en Java
Septima Edición.
México
Pearson Educación, 2008

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

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