Índice

1. Aplicaciones del J2SE SDK1.4.2 de Sun.

Se van a mostrar algunas aplicaciones (hay bastantes más) que se distribuyen con el J2SE de Sun. explicación se utilizará `DIR_JDK` para representar el directorio donde está instalado el JSDK1.4.2, supondré que el código fuente está en un fichero llamado `Clase.java` y que contiene diversas clases: `ClaseAux1, ClaseAux2` y `Clase` siendo ésta última la que contiene el método `main`. Los programas que se citan a continuación se encuentran en el directorio `DIR_JDK\bin`.

1.1. javac

Programa que transforma el código fuente en *bytecode*. El bytecode es el lenguaje que ejecuta la Máquina Virtual de Java (**MVJ**). El compilador de java lee un fichero fuente Java (fichero `.java`) y traduce el este código a bytecode escribiéndolo en un fichero (o ficheros) `.class`. Se generan tantos ficheros `.class` como clases hay en el fichero. Una secuencia de bytecodes representa una serie de instrucciones para la MVJ. Cada instrucción representa un código de operación (*opcode*) de un byte y cero o mas operandos. El opcode indica la acción a realizar, si para esa acción se necesita alguna información ésta está codificada en uno o más operandos que van tras el opcode. Cada opcode tiene un mnemónico con lo que una serie de bytecodes se puede escribir en un lenguaje parecido al ensamblador.

Como ejemplo de compilación:

```
javac Clase.java
```

esto genera tres ficheros:

```
Clase.class ClaseAux1.class ClaseAux2.class
```

1.2. java

Inicia una Máquina Virtual de Java. La MVJ es un computador *abstracto* que ejecuta carga y ejecuta programas Java compilados. La MVJ es *virtual* ya que está implementada en software por encima de una plataforma hardware *real* y un sistema operativo. Por tanto, la MVJ debe estar implementada

en una plataforma particular antes de que los programas se puedan ejecutar sobre esa plataforma. La MVJ proporciona una capa de abstracción entre los programas Java compilados (el bytecode) y la plataforma sobre la que se ejecuta. Esto es importante para asegurar la portabilidad ya que los programas se ejecutan sobre la MVJ, independientemente del hardware sobre el que se está instalada la MVJ.

Para el ejemplo que nos ocupa, llamamos a la MVJ pasándole como argumento el nombre de la clase:

```
java Clase
```

Veamos cuales son los pasos que provoca la orden anterior:

1. Se inicia una MVJ
2. Se carga la representación binaria de la clase (el fichero `.class` con el bytecode). De este proceso se encarga la clase ClassLoader.
3. *Link*, este paso se subdivide en 3:
   - Verificación, asegura que la representación binaria es correcta estructuralmente.
   - Preparación, creación de miembros estáticos y otras optimizaciones.
   - Resolución, la representación binaria contiene referencias a otras clases o interfaces, a sus miembros y métodos y a los constructores. Antes de que una referencia simbólica pueda usarse debe someterse a resolución, donde se comprueba que la referencia es correcta y se reemplaza por una referencia directa.
4. Iniciación
5. Llamada a `Clase.main`, el método main debe ser declarado como público y estático, debe devolver void y acepta como argumento un *array* de `String` (si no se declara de este modo no se podrá ejecutar la aplicación a no ser que se proporcione un cargador de clases adaptado al modo en el que se ha declarado el `main()`)

Todos ficheros `class` a los que se hace referencia en nuestra clase deben ser accesibles por la MVJ (para ser cargados en memoria) cuando se ejecuta la clase. Si se produce algún error durante alguno de esos pasos se lanza una excepción y el proceso se detiene.

1.3. javadoc

Programa que sirve para generar documentación en formato HTML a partir del fichero con el código fuente o a partir de un paquete. El texto necesario para generar esa documentación se extrae de unos comentarios especiales insertados en el código fuente y que documenta las clases, los métodos, etc.

```
javadoc [opciones] Clase
```

Aparte de otras muchas, hay 4 opciones interesantes:

`javadoc -public Clase` muestra sólamente lo público.
`javadoc -protected Clase` muestra lo que está declarado como protected y público.
`javadoc -package Clase` muestra lo declarado con acceso de paquete, protected y publico.
`javadoc -private Clase` muestra todo incluyendo lo privado.

■

1.4. Las que no se explican (por ahora)

Sun proporciona algunas utilidades más entre las que cabe destacar:

- **appletviewer** para visualizar applets.
- **rmiregistry** para crear un registro de objetos remotos.
- **rmic** para generar stubs y skeletons.
- **jar** para comprimir o descomprimir ficheros.
- **keytool** para gestionar claves y certificados.
- **jarsigner** para firmar ficheros jar.

2. Guía de estilo de Java.

La guía de estilo hace referencia a una serie de convenciones en cuando a cómo se deben escribir los programas. Es importante ya que:

- Durante su vida, el software está sujeto a mantenimiento.
- En entornos reales casi nunca una sola persona mantiene el software.
- La convenciones mejoran la legibilidad del código.
- El software es un producto y como tal debe cumplir unos mínimos de calidad.

Sugiero seguir las convenciones que propone Sun (al fin y al cabo fueron ellos los que hicieron el lenguaje).

2.1. Clases

2.1.1. Nombres para las clases

Las clases son abstracciones de entidades, y por tanto los nombres que les asignemos deben estar relacionados con las entidades a las que representan.

En la API (*Application Program Interface*) de Java nos podemos encontrar los siguientes nombres `Vector`, `Array`, `System`, `URL`, `Socket`, `SocketSecurityException` etc...

Como puede verse, los nombres de clases comienzan con mayúscula y si está formado por más de una palabra, el inicio de cada una de ellas está en mayúscula.

2.1.2. Estructura que debe tener una clase.

Parte	*Descripción*
Comentario de documentación para la clase `/**...*/`	*Comentarios que se procesan mediante* `javadoc`.
Sentencia `class`	*Define que se trata de una clase, por supuesto irá precedida de los modificadores pertinentes.*
Variables de clase `static`	*Primero las declaradas como* `public`, *seguidas de las* `protected`, *las de acceso de* `package` *y finalmente las* `private`.
Variables de instancia	*Primero las declaradas como* `public`, *seguidas de las* `protected`, *las de acceso de* `package` *y finalmente las* `private`.
Comentario de documentación para el constructor `/**...*/`	*Comentarios que se procesan mediante* `javadoc`.
Constructores	
Comentario de documentación para el método `/**...*/`	*Comentarios que se procesan mediante* `javadoc`.
Métodos	*Agrupados por funcionalidad, si un método llama a otro lo mejor es que estén lo más cerca posible para que sea más legible.*

2.1.3. Ejemplo

```
import java.math.BigDecimal;

/** Empleado es una clase de ejemplo para mostrar como se pueden documentar las
 *  clases.
 *  La clase Empleado encapsula la informacion necesaria para representar a una persona
 *  que tiene una relación contractual con una entidad.
 *
 * @author Juan
 * @version 1.0
 */
public class Empleado {
   public static final int DIRECCION = 1;
   public static final int MANDO_INTERMEDIO = 2;
   public static final int OPERARIO = 3;

   private String nombre;
   private int edad;
   private BigDecimal sueldo;
   private int puesto;

   public Empleado(String nom, int ed, BigDecimal s, int p){
      nombre = nom;
      edad = ed;
      sueldo = s;
      puesto = p;
   }

   public Empleado(String nom, int ed, BigDecimal s){
      this(nom, ed, s, OPERARIO);
   }

   public void muestraEmpleado(){
      System.out.println("Nombre: " + nombre);
```

```
        System.out.println("Edad: " + edad);
        System.out.println("Sueldo: " + sueldo.setScale(3,BigDecimal.ROUND_HALF_UP));
        System.out.println("Puesto: " + formateaPuesto());
    }

    private String formateaPuesto(){
        String representaPuesto = "";

        switch (puesto){
            case DIRECCION :
                representaPuesto = "Dirección";
                break;
            case MANDO_INTERMEDIO:
                representaPuesto = "Mando intermedio";
                break;
            case OPERARIO:
                representaPuesto = "Operario";
                break;
        }
        return representaPuesto;
    }

}
```

2.2. Constructores

El nombre del constructor es el mismo que el nombre de la clase.

2.2.1. Comentarios de documentación para los constructores.

Cuando el/los constructor/es reciben argumentos estos hay que documentarlos. Para ello se utiliza el *tag* o etiqueta `@param`. Esta etiqueta va seguida del nombre del parámetro (no del tipo), los nombres de los parámetros comienzan con una letra minúscula para distinguirlos de las clases. Tras el nombre del parámetro va una descripción.

2.2.2. Ejemplos

En el programa de ejemplo anterior prodríamos comentar los constructores del siguiente modo.

```
    .
    .
    /** Crea un nuevo Empleado con la informacion suministrada.
      *
      * @param nom nombre del empleado
      * @param ed edad del empleado
      * @param s sueldo del empleado
      * @param p puesto que ocupa en la orgacizacion
      */
    public Empleado(String nom, int ed, BigDecimal s, int p){
    .
    .
    .

    /** Crea un nuevo Empleado con la informacion suministrada asumiendo que
      * el valor por defecto para el puesto es Empleado.OPERARIO
      *
      * @param nom nombre del empleado
      * @param ed edad del empleado
      * @param s sueldo del empleado
      */
    public Empleado(String nom, int ed, BigDecimal s){
    .
    .
    .
```

2.3. Métodos

2.3.1. Nombres para los métodos.

Los métodos representan una acción. Cuando un objeto recibe un mensaje, se le está pidiendo que realice una determinada tarea. Por lo tanto, habitualmente veremos que los nombres de los métodos incluyen un verbo. Como ejemplo del API de Java si tomamos la clase `URL` vemos que los nombres de los métodos son: `getContent()`, `getDefaultPort()`, `getProtocol()`, `openConnection()`, `set(...)`.

Como puede observase, la primera letra es minúscula y si el nombre está formado por más de una palabra, el inicio de las siguientes palabras comienza com mayúscula.

2.3.2. Comentarios de documentación para los métodos.

En los métodos hay que dar una descripción concisa y representativa de lo que realiza el método. Si el método recibe parámetros hay que documentarlos, y si el método devuelve algo que no sea `void` también hay que documentarlo.

- Cada argumento que recibe un método se documenta utilizando la etiqueta `@param` que ya se ha comentado para los constructores.
- Si el método devuelve algo se documenta utilizando la etiqueta `@return` seguida de una descripción.

2.3.3. Ejemplos

En el programa de ejemplo anterior prodríamos comentar los métodos del siguiente modo.

```
.
.
/** Muestra la información sobre el empleado por la consola
  */
public void muestraEmpleado(){
.
.
.
/** Crea un String a partir del <code>int</code> puesto
  * @return un String con el nombre del puesto
  */
private String formateaPuesto(){
.
.
.
```

2.4. Atributos

Los atributos almacenan información relevante para la clase o para los objetos por tanto los nombres que se le asignen deben ser representativos. De otro modo será muy difícil saber que tipo de información almacena el atributo. En el ejemplo propuesto para la clase `Empleado` podríamos haber llamado al atributo `sueldo` simplemente `s`, nos ahorramos tener que escribir pero no contribuye a la claridad del programa que es lo que interesa a largo plazo.

Los atributos también se pueden documentar.

2.5. Sangría

Se propone utilizar un sangrado de 3 espacios tal y como se muestra en el ejemplo. Si se utiliza menos la estructura del programa no se puede ver claramente y si se utiliza más al hacer métodos largos el código se desplaza demasiado hacia la derecha.

2.6. Comentarios a bloques de código

Aparte de los comentarios para documentar el programa, se pueden realizar comentarios a líneas o bloques de código siempre se que esas líneas tengan una cierta complejidad.

Por ejemplo **no** se deben utilizar este tipo de comentarios:

```
.
.
// Asignamos a representaPuesto el String "DIRECCION"
representaPuesto = "DIRECCION";
.
.
.
```

este otro seguramente tiene más sentido:

```
.
.
/*Ya que tenemos puesto representado mediante un int y
deseamos una representación en forma de String
hay que realizar una transformación*/
switch (puesto) {
.
.
.
```