TUTORIAL JAVA

- **Introducción de Java**

Java es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria, la cual es gestionada mediante un recolector de basura.

Se describe a Java como "*Simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico*".

- **Plataformas de java**

Para descargar e instalar java lo puedes hacer desde la página oficial de Oracle. Al instalar java se instala el JRE (Entorno de ejecución de Java) y el JDK (Java Development Kit). El JRE es la implementación de la Máquina virtual de Java que realmente ejecuta los programas de Java y el JDK es un kit de desarrollo de software que se utiliza para escribir aplicaciones con el lenguaje de programación Java. Otro entorno es Java EE el cual es centrado en Java para desarrollar, crear e implementar en línea aplicaciones empresariales basadas en web. Java

- **Herramientas para desarrollo**

El desarrollo y ejecución de aplicaciones en Java exige que las herramientas para compilar (javac.exe) y ejecutar (java.exe).

  - **Javac:** Es una herramienta para leer clases y definiciones de interface escritas en java, y compilarlas al lenguaje de la máquina virtual Java (archivos bytecode).

  - **Java:** Es una herramienta para ejecutar los programas en lenguaje de máquina virtual java.

- **Variables PATH y CLASSPATH**

El desarrollo y ejecución de aplicaciones en Java exige que las herramientas para compilar (javac.exe) y ejecutar (java.exe) se encuentren accesibles, para ello se puede cambiar la variable de entorno **path** y **classpath**, en el sistema operativo windows. La variable **path** contiene la ruta de acceso de los programas que con más frecuencia se use, de modo que para ejecutar un programa bastará con escribir su nombre y Windows lo buscará, de este modo nos ahorramos tener que escribir toda la ruta de directorios hasta llegar a él. La variable **classpath** indica al JDK dónde debe buscar los archivos a compilar o ejecutar, tanto las clases o librerías de Java (el API de Java) como otras clases de usuario, sin tener que escribir en cada ejecución la ruta completa.

Así se ejecutaría el compilador de java sin haber cambiado las variables, en MS-DOS y con el JDK instalado en el directorio C:\Program Files\Java\jdk1.8.0_77 y la clase a compilar de nombre 'miclase.java' en C:\proyectosJava:

C:\Program Files\Java\jdk1.8.0_77\bin\javac C:\proyectosJava\miclase.java

Así se escribiría si se cambian adecuadamente las variables:

javac miclase.java

Variable PATH

PATH es la variable del sistema que utiliza el sistema operativo para buscar los ejecutables necesarios desde la línea de comandos o la ventana Terminal. La variable del sistema PATH se puede establecer utilizando la utilidad de sistema en el panel de control de Windows o en el archivo de inicio del shell en Linux .

Modificación en Windows

Windows 7
Para saber que valores contiene la variable path escribiremos lo siguiente en la consola de MS-DOS:

C:>path

1. Para saber si están configuradas las herramientas java y javac y ejecute los siguientes comandos:

java –version

javac -version

Si aparece un error de java: Comando no encontrado, esto indica que la variable path no está configurada correctamente.

2. Seleccione Equipo en el menú Inicio
3. Seleccione Propiedades del sistema en el menú contextual
4. Haga clic en Configuración avanzada del sistema > ficha Opciones avanzadas
5. Haga clic en Variables de entorno, en Variables del sistema, busque **PATH** y haga clic en él.
6. En las ventanas Editar, modifique **PATH** agregando la ubicación de la clase al valor de **PATH**. Si no dispone del elemento **PATH**, puede optar por agregar una nueva variable y agregar **PATH**como el nombre y la ubicación de la clase como valor.
7. En el final de la línea escriba la ruta en la cual se encuentran los programas java y y javac, por ejemplo “C:\Program Files\Java\jdk1.8.0_77\bin\”, cada ruta de la variable PATH va separada con punto y coma.
8. Vuelva a abrir la ventana del indicador de comandos y ejecute los siguientes comandos:

```
C:\Users\Edil>java -version
java version "1.8.0_77"
Java(TM) SE Runtime Environment (build 1.8.0_77-b03)
Java HotSpot(TM) Client VM (build 25.77-b03, mixed mode)

C:\Users\Edil>javac -version
javac 1.8.0_77
```

Si aparece un error de java: Comando no encontrado, esto indica que la variable path no está configurada correctamente.

Para saber qué valores contiene el CLASSPATH (no contiene ninguno por defecto) bastará escribir en la consola MS-DOS:

C:>set

- Variable CLASSPATH

Para saber qué valores contiene el CLASSPATH (no contiene ninguno por defecto) bastará escribir en la consola MS-DOS:

C:>set

y aparecerá la lista de variables de entorno con sus correspondientes valores. Si no aparece la variable CLASSPATH quiere decir que ésta no contiene ningún valor. Para asignarle un valor teclearemos lo siguiente en una ventana de MS-DOS:

C:>set CLASSPATH=C:\MisClasesDeJava

Siendo *MisClasesDeJava* el directorio donde tenemos nuestras clases de Java. Para introducir varios directorios los separaremos por punto y coma:

C:>set CLASSPATH=C:\MisClasesDeJava;C:\MisOtrasClasesDeJava

Finalmente si la variable CLASSPATH ya contiene valores y queremos añadir más tendremos que hacerlo del siguiente modo:

C:>set CLASSPATH= C:\MisClasesDeJava;%CLASSPATH%

Una forma general de indicar estas dos variables es crear un fichero batch de MS-DOS donde se indiquen los valores de dichas variables. Cada vez que se abra una ventana de MS-DOS será necesario ejecutar este fichero *.bat para asignar adecuadamente estos valores. Un posible fichero llamado *variablesEntorno.bat*, teniendo en cuenta que el JDK está en el directorio C:\Program Files\Java\jdk1.8.0_77, podría ser como sigue:

```
@echo OFF
set JAVA_HOME=C:\Program Files\Java\jdk1.8.0_77
set CLASSPATH=.;%JAVA_HOME%\lib\
echo ----------------------------
echo Variable claspath fijada
echo ----------------------------
set PATH=%JAVA_HOME%\bin;%PATH%
echo ----------------------------
echo Variable path fijada
echo ----------------------------
```

Si está configurando una versión Linux diríjase al siguiente link, https://www.java.com/es/download/help/path.xml, en la opción configurar establecimiento de la variable Path en Linux.

Estructura general de un programa en java

Se cuenta con una clase que contiene el programa principal (aquella que contiene la función *main ()*) y algunas clases de usuario (las especificas de la aplicación que se esta desarrollando) que son usadas por el programa principal. Los ficheros tienen extensión ***.java*** mientras que los ficheros compilados tienen la extensión ***.class.***

Un fichero fuente (***.java***) puede contener mas de una clase, pero solo una puede ser ***public***.
El nombre del fichero fuente debe coincidir con el de la clase ***public*** (con la extensión ***.java***).

Si en un fichero aparece la declaración (***public class MiClase{ … })*** Entonces el nombre del fichero deberá ser ***MiClase.java.*** (Teniendo en cuenta las mayúsculas y minúsculas)
Si la clase no es ***public*** no es necesario que su nombre coincida con el del fichero.

Una aplicación esta constituida por varios ficheros ***.class***. Cada clase realiza unas funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. La aplicación se ejecuta por medio del nombre de la clase que contiene la función *main ()* (sin la extensión ***.class***).

Ejemplo:

Cree la siguiente clase

prueba.java

```
public class prueba{
    public static void main(String[] args) {
      System.out.println("Hola mundo");
    }
}
```

Compile el archivo mediante el comando

javac prueba.java

Y ejecútela mediante el comando

java prueba

- **Concepto de Clase**

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

```
[public] class Classname{
     //definición de variables y métodos
     …
}
```

Donde la palabra **public** es opcional: si no se pone, la clase tiene la visibilidad por defecto, esto es, sólo es visible para las demás clases del **package**. Todos los métodos y variables deben ser definidos dentro del bloque {…} de la clase.

Un objeto (en inglés, instance) es un ejemplar concreto de una clase. Las clases son como tipos de variables, mientras que los objetos son como variables concretas de un tipo determinado.

Classname unObjeto;

Classname otroObjeto;

A continuación se enumeran algunas características importantes de las clases:

1. Todas las variables y funciones de Java deben pertenecer a una clase. No hay variables y funciones globales.
2. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase public. Este fichero se debe llamar como la clase public que contiene con extensión *.java. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.

3. Si una clase contenida en un fichero no es public, no es necesario que el fichero se llame como la clase.
4. Los métodos de una clase pueden referirse de modo global al objeto de esa clase al que se aplican por medio de la referencia **this**.

- **Variables miembro**

Las variables miembros de una clase o campos pueden ser de tipos primitivos (boolean, int, long, double, etc.) o referencias a objetos de otra clase (composición).

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembro de tipos primitivos se inicializan siempre de modo automático (*false* para boolean, el *carácter nulo* para char y *cero* para los tipos numéricos), incluso antes de llamar al constructor. Sin embargo, lo más adecuado es inicializarlas también en el constructor o también pueden inicializarse explícitamente en la declaración.

Las variables miembro pueden ir precedidas en su declaración por uno de los modificadores de acceso: **public**, **private**, **protected** y **package** (que es el valor por defecto y puede omitirse); junto con los modificadores de acceso de la clase (**public** y **package**), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables miembro.

Las variables miembro de un objeto concreto se acceden directamente, o mediante los métodos miembros.

objeto.campo

- **Métodos (Funciones Miembro)**

  - **Métodos de objeto**

Son funciones definidas dentro de una clase. Los métodos ***static*** o de clase, se aplican siempre a un objeto de la clase por medio del operador punto (**.**). Los métodos pueden tener otros argumentos explícitos que van entre paréntesis.

```
public Circulo elMayor (Circulo c){    //header y comienzo del método
      if (this.r>=c.r)                    //body
            return this;                  //body
      else                                //body
            return c;                     //body
}                                         //final del método
```

El header consta del cualificador de acceso, del tipo del valor de retorno, del nombre de la función y de una lista de argumentos explícitos si los hay entre paréntesis.

Los métodos tienen visibilidad directa de las variables miembro del objeto que es su argumento implícito, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto.

El valor de retorno puede ser un valor de un tipo primitivo o una referencia.

Los métodos pueden definir variables locales, en las cuales su visibilidad llega desde la definición al final del bloque en el que han sido definidas.

- **Métodos sobrecargados (overloaded)**

  Son métodos distintos que tienen el mismo nombre, pero que se diferencian por el número y/o tipo de los argumentos.

  Para llamar un método sobrecargado, Java sigue unas reglas para determinar el método concreto que debe llamar:

  - Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada, se llama ese método.
  - Si no existe un método que se ajuste, se promueve los argumentos actuales al tipo inmediatamente superior (por ejemplo: **char** a **int**, **int** a **long**, **float** a **double**, etc) y se llama al método correspondiente.
  - Si solo existen métodos con argumentos de un tipo restringido (por ejemplo, **int** en vez de **long**) se debe hacer un ***cast*** explicito en la llamada.
  - El valor de retorno no influye en la elección del método sobrecargado. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre que solo difieran en el valor de retorno.

- **Paso de argumento a métodos**

Los argumentos de los tipos primitivos se pasan por valor. El método recibe una copia del argumento actual; si se modifica esta copia, el original no queda modificado. La forma de modificar dentro de un método una variable de un tipo primitivo es incluirla como variable miembro en una clase y pasar como argumento una referencia a un objeto de dicha clase. Las *referencias* se pasan también por *valor*, pero a través de ellas se puede modificar los objetos referenciados.

En Java no se pueden pasar métodos como argumentos a otros métodos. Se puede pasar por referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.

Si un método devuelve ***this*** (es decir, un objeto de la clase) o una referencia a otro objeto, ese objeto puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así sucesivamente. En este caso aparecerán varios métodos en la misma sentencia unidos por el operador punto (.).

```
String numeroComoString = "8.9.78";
```

```
Float p = Float.valueOf(numeroComoString).floatValue();
```

Donde el método **`valueOf(String)`** de la clase ***java.lang.Float*** devuelve un objeto de la clase ***Float*** sobre el que se aplica el método ***floatValue()***, que finalmente devuelve una variable primitiva de tipo ***float.*** El ejemplo anterior se podría desdoblar en las siguientes sentencias:

```
String numeroComoString = "8.9.78";
Float f = Float.valueOf(numeroComoString);
float p = f.floatValue();
```

Se pueden encadenar varias llamadas a métodos por medio del operador punto (.).

- **Constructores**

Java no permite que hayan variables miembro que no estén inicializadas. Java siempre inicializa con valores por defecto las variables miembro de clases y objetos. El segundo paso en la inicialización correcta de objetos es el uso de constructores.

Un constructor es un método que se llama automáticamente cada vez que se crea un objeto de una clase. Su misión es reservar memoria e inicializar las variables miembro de la clase; no tienen valor de retorno y su nombre es el mismo que el de la clase, su argumento implícito es el objeto que se está creando.

Una clase tiene varios constructores, que se diferencian por el tipo y numero de sus argumentos. El ***constructor por defecto*** no tiene argumentos, el programados debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un constructor de una clase puede llamar a otro constructor previamente definido en la misma clase por medio de la palabra ***this***.

- **Destrucción de objetos**

El sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han perdido la referencia, es decir, los objetos que ya no tienen ningún nombre que permita acceder a ellos, por ejemplo por haber llegado al final del bloque en el que se habían definidos, porque la referencia se le ha asignado el valor de ***null*** o porque se le ha asignado la dirección de otro objeto. A esta característica se le llama ***garbage collection*** (recogida de basura).

- **Concepto de package**

Es una agrupación de clases. En la API de Java 1.1 había 22 packages; en Java 1.2 hay 59 packages. Teniendo en cuenta que el usuario puede también crear sus propios packages.

Para que una clase pase a formar parte de un package llamado ***pkgName***, hay q introducir en ella la sentencia:

package **pkgName**;

La cual debe ser la primera sentencia del fichero.
Todas las clases que forman parte de un package deben estar en el mismo directorio.

Son usados con la finalidad de:
1) Agrupar clases relacionadas.
2) Evitar conflictos de nombres.
3) Para ayudar en el control de la accesibilidad de clases y miembros.

- **Herencia**

Con la herencia se puede construir una clase a partir de otra, para indicar que una clase deriva de otra se utiliza la palabra **extends**.

Ejemplo:

class CirculoGrafico **extends** Circulo {…}

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser redefinidas (overridden) en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la *subclase* contuviera un objeto de la *superclase*; en realidad lo "amplia" con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de Java creadas por el programador tienen una superclase. Cuando no se indica explícitamente una superclase con la palabra extends, la clase deriva de **java.lang.Object**, que es la clase raíz de toda la jerarquía de clases de Java. Como consecuencia, todas las clases tienen algunos métodos que han heredado de **Object**.

- **Redefinición de métodos heredados:**

Una clase puede redefinir cualquiera de los métodos heredados de su superclase que no sean final (en la definición de un método sirve para indicar que no puede ser sobrescrito por las subclases). El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Los métodos de la superclase que han sido redefinidos pueden ser todavía accedidos por medio de la palabra **super** desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases; se puede incluso llamar a un constructor de una superclase, usando la sentencia **super()**. Ejemplo:

```
public class vehiculo{
       double velocidad;
       ...
       public void acelerar(double cantidad){
            velocidad+=cantidad;
       }
}

public class coche extends vehiculo{
       double gasolina;
       public void acelerar(double cantidad){
            super.acelerar(cantidad);
            gasolina*=0.9;
       }
}
```

En el ejemplo anterior, la llamada ***super.acelerar(cantidad)*** llama al método acelerar de la clase vehículo. Es necesario redefinir el método acelerar en la clase coche ya que aunque la velocidad varia igual que en la superclase, hay que tener en cuenta el consumo de gasolina.

Los métodos redefinidos pueden ampliar los derechos de acceso de la superclase (por ejemplo ser **public**, en lugar de **protected** o **package**), pero nunca restringirlos. Los métodos de clase o **static** no pueden ser redefinidos en las clases derivadas.

- **Concepto de interfaces**

Una interface es un conjunto de declaraciones de métodos (sin definición). Puede definir constantes, que son implícitamente ***public, static*** y ***final***, y deben inicializarse en la declaración. Todas las clases que implementan una determinada interface están obligadas a proporcionar una definición de los métodos de la interface, y en ese sentido adquieren una conducta o modo de funcionamiento.

Una clase puede implementar una o varias interfaces. Para indicar que una clase implementa una o mas interfaces se ponen los nombres de las interfaces, separas por comas, detrás de la palabra **implements**, va a la derecha del nombre de la clase o del nombre de la super-clase en el caso de la herencia.

```
puclic class CirculoGrafico extends Circulo
                            implements Dibujable, Cloneable{
.....
}
```

- **Definición de interfaces.**

Una interface se define de un modo muy similar a las clases. Se presenta un ejemplo de la interface ***Dibujable.***

```
//fichero Dibujable.java

import java.awt.Graphics;
public interface Dibujable {
    public void setPosicion (double x, double y);
    public void dibujar (Graphics dw);
}
```

Cada interface ***public*** debe ser definida en un fichero ***.java** con el mismo nombre de la interfaz.

- **Utilización de interfaces.**

Las constantes definidas en una interface se pueden utilizar en cualquier clase (aunque no implemente la interface) precediéndolas del nombre de la interface, como por ejemplo (Suponiendo que **PI** hubiera sido definida en **Dibujable**):

```
area = 2.0 + dibujable.PI*r;
```

Sin embargo, en las clases que implementan la interface las constantes se pueden utilizar directamente, como si fueran constantes de la clase.

- **Excepciones**

En Java, una **Excepción** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa.

- Lanzar una Excepción:

Cuando en un método se produce una situación anómala es necesario lanzar una excepción. El proceso de lanzamiento de una excepción es el siguiente:

1. Se crea un objeto **Exception** de la clase adecuada.
2. Se lanza la excepción con la sentencia **throw** seguida del objeto **Exception** creado.

```
//Código que lanza la excepción MyException una vez detectado el error.
MyException me = new MyException ("MyException message").
throw me;
```

Esta excepción deberá ser capturada (**catch**) y gestionada en el propio método o en algún otro lugar del programa.

Al lanzar una excepción el método termina de inmediato, sin devolver ningún valor. Solamente en el caso de que el método incluya los bloques **try/catch/finally** se ejecutara el bloque **cath** que la captura o el bloque **finally** (si existe).

```
public void leerFichero(String fich) throws EOFException,
FileNotFoundException {…}
```

Se puede poner únicamente una superclase de excepciones para indicar que se pueden lanzar excepciones de cualquiera de sus clases derivadas. En el caso anterior seria equivalente a:

```
public void leerFichero(String fich) throws IOException {…}
```

Las excepciones pueden ser lanzadas directamente por `leerFichero()` o por alguno de los métodos llamados por `leerFichero()`, ya que las clases `EOFException` y `FileNotFoundException` derivan de `IOException`.

- **Capturar una Exception:**

Cuando se lanza una excepción se debe:
1. Gestionar la excepción con una construcción del tipo **try**{…} **catch**{…}.
2. Relanzar la excepción hacia un método anterior en el stack, declarando que su método también lanza dicha excepción, utilizando para ello la construcción **throws** en el encabezado del método.

- **Bloques try y catch:**

El código dentro del bloque **try** esta "vigilado". Si se produce una situación anormal y se lanza por lo tanto una excepción el control salta o sale del bloque **try** y pasa al bloque **catch**, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques **catch** como sean necesarios, cada uno de los cuales tratara un tipo de excepción. Las excepciones se pueden capturar individualmente o en grupo, por medio de una superclase de la que deriven todas ellas.

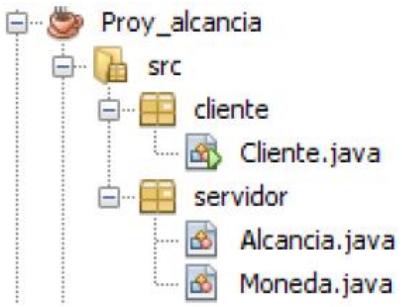
El bloque **finally** es opcional. Si se incluye sus sentencias se ejecutan siempre, sea cual sea la excepción que se produzca o si no se produce ninguna. El bloque **finally** se ejecuta aunque en el bloque try haya un **return**.

En el siguiente ejemplo se presenta un método que debe "controlar" una **IOException** relacionada con la lectura ficheros y una **MyException** propia:

```
void metodo1(){
   try{
      //Código que puede lanzar las excepciones IOException y MyException
      }catch(IOException e1){//Se ocupa de IOException simplemente dando aviso
            System.out.println(e1.getMessage()};
         }catch(MyException e2){
               //Se ocupa de MyException dando un aviso y finalizando la función
            System.out.println(e2.getMessage()}; return;
         }finally{  //Sentencias que se ejecutaran en cualquier caso
            ...
         }
        ...
  }  //Fin del metodo1
```

- **Ejemplo Práctico**

Con el archivo dir_repaso.bat entregado, la distribución de archivos quedaría de la siguiente manera:



Donde el archivo *Cliente.java* contendrá el siguiente código:

```
package cliente;

import java.io.IOException;
import servidor.Alcancia;
import servidor.Moneda;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Cliente {

public static void main(String[] args) throws IOException {
      int opcion = 0;
      String linea = "";
      Alcancia a=new Alcancia();
```

```
        Moneda mon=new Moneda(20);

    do{
        System.out.println("|*****************MENU*******************|");
        System.out.println("|1. Ingresar moneda de 20 |");
        System.out.println("|2. Total de la Alcancia |");
        System.out.println("|3. Estado de la Alcancia |");
        System.out.println("|4. Salir |");
        System.out.println("|****************************************|");
        System.out.println("Ingrese la opcion: ");

        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        linea = br.readLine();
        opcion = Integer.parseInt(linea);
        switch(opcion){
        case 1:
              a.ingresarMoneda(mon);
              System.out.println("Se ha ingresado la moneda de 20");
        break;
        case 2:
              System.out.println("La alcancia tiene: "+a.valorAhorrado()+"
        pesos");
        break;
        case 3:
              System.out.println("de 20: "+a.cantidadDenom(20));
        break;
        case 4:
              a.romperAlcancia();
        break;
        default:
        break;
        }
     }while(opcion!=4);
    }
}
```

El archivo *Moneda.java* contendrá el siguiente código:

```
package servidor;

public class Moneda {
      //Atributos
      private int denom;
      private int cantidad;
      //Metodos
      public Moneda(int denom){
      this.denom=denom;
      cantidad=0;
      }
      public int getDenom() {
      return denom;
      }
      public void setDenom(int denom) {
```

```
		this.denom = denom;
		}
		public int getCantidad() {
		return cantidad;
		}
		public void setCantidad(int cantidad){
			this.cantidad=cantidad;
		}
		public void acumularCantidad(){
		cantidad=cantidad+1;
		}
}
```

El archivo *Alcancia.java* contendrá el siguiente código:

```
package servidor;

public class Alcancia{

		private Moneda mon20;
		private int ahorrado;
		public Alcancia(){
		mon20=new Moneda(20);
		}
		public void ingresarMoneda(Moneda m) {
			switch (m.getDenom()){
				case 20 : {mon20.acumularCantidad(); break;}
			}
			ahorrado=this.valorAhorrado();
		}
		public int valorAhorrado() {
			ahorrado=20*mon20.getCantidad();
			return ahorrado;
		}
		public void romperAlcancia() {
			mon20=null;
		}
		public int cantidadDenom(int denom){
			int cantidad=0;
			switch(denom){
				case 20 : {cantidad=mon20.getCantidad(); break;}
			}
			return cantidad;
		}
		public int getAhorrado() {
			return ahorrado;
		}
}
```

Compilación del Programa:

- Se abre una terminal, ubicados en el directorio *src* y se escribe el siguiente comando:

javac –d ..\bin cliente\Cliente.java

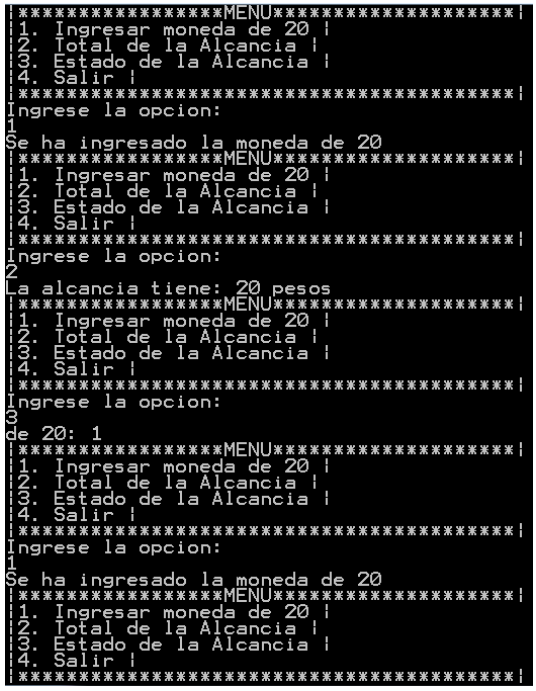
- Si el programa no tiene errores, la compilación generara un nuevo fichero llamado ***Cliente.class,*** el cual contiene el código compilado; denominado objeto de nuestro programa. El código objeto es el que se pasa a la JVM para que lo ejecute.

Ejecución del Programa:

Para ejecutar el programa, una vez realizada con éxito la compilación, nos ubicamos en el directorio *bin* y digitamos el siguiente comando:

java cliente.Cliente

El resultado debe ser que veamos en nuestra ventana de MS-DOS o terminal el menú del Programa:



Práctica:
Partiendo del ejemplo anterior se debe modificar las opciones:
1. En la clase Moneda almacenar el tipo de moneda (pesos o euros).
2. Diferentes opciones para ingresar y retirar monedas de 50 y 100 respectivamente, para únicamente dos tipos de moneda (pesos, euros).
3. Total de la Alcancía donde se sumen las monedas ingresadas, y según el tipo de moneda.
4. Mostrar la cantidad de monedas que hay por cada denominación (número de monedas de 50 y 100), según el tipo de moneda.
Debe manejar excepciones al retirar las monedas si la alcancía no tiene dinero suficiente.