Departamento de Ingeniería Telemática
Escuela Superior de Ingenieros
Universidad de Sevilla

PRACTICA 5: Encapsulación y Utilidades.

1. OBJETIVO

La programación orientada a objetos es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones. Está basado en varias técnicas, incluyendo herencia, abstracción, polimorfismo y encapsulación. En esta práctica se hace especial énfasis en los aspectos de **encapsulación**. El objetivo de esta práctica es también presentar la clase `String` y las clases envolventes (también conocidos como *wrappers*) de los tipos básicos de Java. Se presentan los métodos de estas clases, y se proponen ejercicios.

2. ENCAPSULACIÓN

El **encapsulado** es el mecanismo que permite agrupar el código y los datos manipulados, y que mantiene a ambos alejados de posibles interferencias o usos indebidos. El encapsulado es como un envoltorio protector que evita que otro código que está fuera pueda acceder arbitrariamente al código o a los datos. El acceso al código y a los datos se realiza de forma controlada a través de una interfaz bien definida.

De esta forma el usuario de una clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos.

Formas de encapsular: La clase

En Java la base del encapsulado es la **clase**. Una clase define la estructura y el comportamiento (datos y código) que serán compartidos por un conjunto de objetos. Cada objeto de una clase dada contiene la estructura y el comportamiento definidos por la clase, como si fuera grabado por un molde con la forma de clase. Por esta razón, a los objetos se les llama **instancias de una clase**. Una clase es una construcción lógica; un objeto tiene realidad física. Cuando se crea una clase, hay que especificar el código y los datos que constituyen esa clase. En su conjunto, estos elementos son **miembros** de la clase. Los datos definidos en la clase son las **variables de instancia** o **atributos**. El código que opera sobre esos datos se conoce como **métodos** de la clase.

Dado que el propósito de una clase es encapsular la complejidad, hay mecanismos para ocultar la complejidad de la implementación dentro de la clase. Cada método o variable de una clase se puede marcar como **privado** o **público**. La interfaz **pública** de una clase representa todo lo que los usuarios externos de la clase necesitan conocer, o que pueden conocer. Los métodos y datos **privados** sólo pueden ser utilizados por código miembro de la clase. Cualquier otro código que no es miembro de la clase no puede acceder a un método o variable privado. Dado que los métodos privados de una clase sólo pueden ser utilizados por otras partes del programa a través

de los métodos públicos de la clase, puede asegurar que no se van a producir accesos no permitidos. Esto significa que la interfaz pública debería ser diseñada cuidadosamente para no exponer demasiado los trabajos internos de una clase.

En el siguiente ejemplo se muestra una clase llamada `ClaseBase`, con atributos que contienen los cuatro niveles de acceso (**`private`**, **`protected`**, paquete y **`public`**). En este punto se hace especial énfasis en los niveles de acceso privado (**`private`**) y público (**`public`**). Los otros niveles de acceso se verán en los siguientes apartados.

```
package fp2.poo.practica5.p1;

public class ClaseBase {

    private   int atributoPrivado;
    protected int atributoProtegido;
              int atributoPaquete;
    public    int atributoPublico;

    public ClaseBase() {
        System.out.println("_____________________ClaseBase: Constructor_____________________");
        this.atributoPrivado = 1;
        System.out.println("Constructor de ClaseBase, atributo privado   "  + this.atributoPrivado);
        this.atributoProtegido = 2;
        System.out.println("Constructor de ClaseBase, atributo protegido "  + this.atributoProtegido);
        this.atributoPaquete = 3;
        System.out.println("Constructor de ClaseBase, atributo paquete   "  + this.atributoPaquete);
        this.atributoPublico = 4;
        System.out.println("Constructor de ClaseBase, atributo publico   "  + this.atributoPublico);
        System.out.println("_____________________________________________________________________");
    }

    private void metodoPrivado() {
        System.out.println("ClaseBase: Metodo privado\n");
    }

    protected void metodoProtegido() {
        System.out.println("ClaseBase: Metodo protegido\n");
    }

    void metodoPaquete() {
        System.out.println("ClaseBase: Metodo acceso en el paquete\n");
    }

    public void metodoPublico () {
        System.out.println("ClaseBase: Metodo publico\n");
    }
}
```

ClaseBase.java

```
package fp2.poo.practica4.p1;

import fp2.poo.practica4.p1.ClaseBase;

public class Main {

    public static void main(String args[]){
        ClaseBase b = new ClaseBase();
    }
}
```

Main.java

Descargue el código de la plataforma virtual para realizar al práctica 5 y compile las dos clases anteriores mediante el siguiente comando:

`make -f make-5 compilaJava01`

Ejecute el ejemplo anterior

`make -f make-5  ejecuta01`

Incluya en el método principal llamadas al método privado y al método público de la **`ClaseBase`** y acceda a los atributos privado y público de la **`ClaseBase`**. Observe e interprete el resultado de la compilación.

La encapsulación aplicando herencia

La herencia es el proceso mediante el cual una clase de objetos (clase derivada o *subclase*) adquiere las propiedades de otra clase de objetos (clase base o *superclase*). De esta se consigue organizar de forma jerárquica las clases. La herencia permite la reutilización y extensibilidad del software, ya que La clase derivada puede reutilizar el código de la superclase. (En Java, para que una clase derive de otra se utiliza la palabra reservada **`extends`**).

En Java el nivel de acceso protegido (**`protected`**) se utiliza para poder acceder a miembros de la superclase.

La encapsulación aplicando paquetes

Un **paquete** en Java es un contenedor de clases que permite agrupar las distintas partes de un programa cuya funcionalidad tienen elementos comunes. Todas las clases e interfaces en el lenguaje de programación Java van incorporadas en un paquete.

En Java el acceso a nivel de paquete es el nivel por defecto. Todos los miembros de un paquete, cuyo nivel de acceso es a nivel de paquete, pueden ser accedidos desde cualquier parte del propio paquete, pero no pueden ser accedidos desde fuera del paquete.

Todos los miembros de un paquete cuyo nivel de acceso es protegido, pueden ser accedidos desde cualquier parte del propio paquete, pero desde fuera del paquete sólo pueden ser accedidos en las clases derivadas (mediante el uso de herencia).

Ejemplo de encapsulación aplicando herencia en el mismo paquete

En el siguiente ejemplo se proporcionan las siguientes clases:

1. `ClaseBase`, que pertenece al paquete `fp2.poo.practica5.p1`.
2. `ClaseDerivadaMismoPaquete`, que pertenece al mismo paquete que `ClaseBase` (`fp2.poo.practica5.p1`), y deriva de `ClaseBase`.
3. `Main`, que pertenece al mismo paquete que `ClaseBase` y `ClaseDerivadaMismoPaquete` (`fp2.poo.practica5.p1`), en la que se instanciarán objetos de las dos clases anteriores.

```
package fp2.poo.practica5.p1;

import fp2.poo.practica5.p1.ClaseBase;

public class ClaseDerivadaMismoPaquete extends fp2.poo.practica5.p1.ClaseBase {

    public ClaseDerivadaMismoPaquete () {
        //this.atributoPrivado = 1;
        //System.out.println("Constructor de ClaseDerivadaMismoPaquete , atributo privado   "
        //                   + atributoPrivado);
        this.atributoProtegido = 2;
        System.out.println("Constructor de ClaseDerivadaMismoPaquete , atributo protegido "
                           + atributoProtegido);
        this.atributoPaquete = 3;
        System.out.println("Constructor de ClaseDerivadaMismoPaquete , atributo paquete   "
                           + atributoPaquete);
        this.atributoPublico= 4;
        System.out.println("Constructor de ClaseDerivadaMismoPaquete , atributo publico   "
                           + atributoPublico);

    }

    private void metodoDerivPaqPrivado () {
        System.out.println("Metodo privado en ClaseDerivadaMismoPaquete \n");
    }

    protected void metodoDerivPaqProtegido () {
        System.out.println("Metodo protegido en ClaseDerivadaMismoPaquete \n");
    }

    void metodoDerivPaqPaquete () {
        System.out.println("Metodo acceso en el paquete en ClaseDerivadaMismoPaquete \n");
    }

    public void metodoDerivPaqPublico () {
        System.out.println("Metodo publico en en ClaseDerivadaMismoPaquete \n");
    }
}
```

`ClaseDerivadaMismoPaquete.java`

Quite los comentarios al código del fichero `Main.java`, compile y ejecute

Compile y ejecute mediante los siguientes comandos:

`make -f make-5 compilaJava02`

`make -f make-5  ejecuta02`

Observe que en la construcción del objeto de la subclase `ClaseDerivadaMismoPaquete`, primero se construye la parte correspondiente a la superclase y luego la correspondiente a la subclase. Por tanto, en este caso primero se ejecuta el código del constructor implementado en la `ClaseBase`, y luego el código incluido en el constructor de la subclase `ClaseDerivadaMismoPaquete`.

Elimine los comentarios en el constructor de la subclase `ClaseDerivadaMismoPaquete`, y compruebe que el atributo privado no es accesible desde la clase derivada (`ClaseDerivadaMismoPaquete`). Observe que los demás atributos pueden ser accedidos desde la clase (`ClaseDerivadaMismoPaquete`).

Incluya en el método principal llamadas a todos los métodos de la `ClaseDerivadaMismoPaquete`, incluyendo a los que hereda de **`ClaseBase`**. Y compruebe cuáles de ellos son accesibles.

Ejemplo de encapsulación en otro paquete

En el siguiente ejemplo se proporcionan las siguientes clases:

1. `ClaseBase`, que pertenece al paquete `fp2.poo.practica5.p1`.
2. `ClaseSinHerenciaEnOtroPaquete`, que pertenece a un paquete diferente (`fp2.poo.practica5.p2`) que `ClaseBase` e instancia en su constructor un objeto del tipo `ClaseBase`.
3. `Main`, que pertenece a un paquete diferente que `ClaseBase` y al mismo paquete que `ClaseSinHerenciaEnOtroPaquete` (`fp2.poo.practica5.p2`), en la que se instanciará un objeto de la clase `ClaseSinHerenciaEnOtroPaquete`.

```
package fp2.poo.practica5.p2;

import fp2.poo.practica5.p1.ClaseBase;

public class ClaseSinHerenciaEnOtroPaquete {


    public ClaseSinHerenciaEnOtroPaquete() {
        ClaseBase obj = new ClaseBase();
        System.out.println();
        System.out.println("_ ClaseSinHerenciaEnOtroPaquete: Constructor_");

        //obj.atributoPrivado= 1;
        //System.out.println("Constructor de ClaseBase, atributo privado   "
        //                   + obj.atributoPrivado);

        //obj.atributoProtegido= 2;
        //System.out.println("Constructor de ClaseBase, atributo protegido "
        //                     + obj.atributoProtegido);

        //obj.atributoPaquete= 3;
        //System.out.println("Constructor de ClaseBase, atributo paquete   "
        //                   + obj.atributoPaquete);

        obj.atributoPublico= 4;
        System.out.println("Constructor de ClaseBase, atributo publico   "
                           + obj.atributoPublico);
        System.out.println("_____________________________________________");
    }
}
```

`ClaseSinHerenciaEnOtroPaquete.java`

```
package fp2.poo.practica5.p2;

//import fp2.poo.practica5.p2.ClaseConHerenciaEnOtroPaquete;
import fp2.poo.practica5.p2.ClaseSinHerenciaEnOtroPaquete ;

public class Main {

    public static void main(String args[]){
        //ClaseConHerenciaEnOtroPaquete b = new ClaseConHerenciaEnOtroPaquete();
        ClaseSinHerenciaEnOtroPaquete c = new ClaseSinHerenciaEnOtroPaquete();
    }
}
```

`Main.java`

Compile y ejecute mediante los siguientes comandos:

make -f make-5 compilaJava03

make -f make-5 ejecuta03

Observe que la clase `ClaseSinHerenciaEnOtroPaquete` solamente puede acceder a los miembros públicos del objeto del tipo `ClaseBase`.

Quite los comentarios del constructor de la clase `ClaseSinHerenciaEnOtroPaquete`, para poder observar lo que ocurre al acceder a los demás miembros del objeto del tipo `ClaseBase`.

Ejemplo de encapsulación aplicando herencia en otro paquete

En el siguiente ejemplo se proporcionan las siguientes clases:

1. `ClaseBase`, que pertenece al paquete `fp2.poo.practica5.p1`.
2. `ClaseConHerenciaEnOtroPaquete`, que pertenece a un paquete diferente (`fp2.poo.practica5.p2`) que `ClaseBase` y deriva de `ClaseBase`.
3. `Main`, que pertenece a un paquete diferente que `ClaseBase` y al mismo paquete que `ClaseConHerenciaEnOtroPaquete` (`fp2.poo.practica5.p2`), en la que se instanciará un objeto de la clase `ClaseSinHerenciaEnOtroPaquete` y `ClaseConHerenciaEnOtroPaquete`.

```
package fp2.poo.practica5.p2;

import fp2.poo.practica5.p1.ClaseBase;

public class ClaseConHerenciaEnOtroPaquete extends ClaseBase {


    public ClaseConHerenciaEnOtroPaquete() {
        System.out.println("__ClaseConHerenciaEnOtroPaquete: Constructor____");
        //this.atributoPrivado= 1;
        //System.out.println("Constructor de ClaseBase, atributo privado   "
        //                   + this.atributoPrivado);

        this.atributoProtegido= 2;
        System.out.println("Constructor de ClaseBase, atributo protegido "
                           + this.atributoProtegido);

        //this.atributoPaquete= 3;
        //System.out.println("Constructor de ClaseBase, atributo paquete   "
        //                   + this.atributoPaquete);

        this.atributoPublico= 4;
        System.out.println("Constructor de ClaseBase, atributo publico   "
                           + this.atributoPublico);
        System.out.println("_____________________________________________________");
    }
}
```

`ClaseConHerenciaEnOtroPaquete.java`

```
package fp2.poo.practica5.p2;
```

```
import fp2.poo.practica5.p2.ClaseConHerenciaEnOtroPaquete;
import fp2.poo.practica5.p2.ClaseSinHerenciaEnOtroPaquete ;

public class Main {

    public static void main(String args[]){
        ClaseConHerenciaEnOtroPaquete b = new ClaseConHerenciaEnOtroPaquete();
        ClaseSinHerenciaEnOtroPaquete c = new ClaseSinHerenciaEnOtroPaquete();
    }
}
```

Main.java

Antes de compilar debe quitar los comentarios de la clase **`Main`**, del paquete `fp2.poo.practica5.p2`. Compile y ejecute mediante los siguientes comandos:

`make -f make-5 compilaJava04`

`make -f make-5  ejecuta04`

Observe que la clase `ClaseConHerenciaEnOtroPaquete` puede acceder a los miembros públicos y protegidos de la superclase `ClaseBase`.

Quite los comentarios del constructor de la clase `ClaseConHerenciaEnOtroPaquete`, para poder observar lo que ocurre al acceder a los demás miembros del objeto del tipo `ClaseBase`.

3. CADENAS

Para esta parte se utilizará el código descargado de la plataforma virtual, pero en este caso utilizando el makefile con el nombre **`makeP5`**. Para compilar todos los ejemplos correspondientes a esta parte ejecute:

```
make –f makeP5 compilaJava
```

Una cadena es una secuencia de caracteres. En C, las cadenas se implementan como matrices de caracteres terminadas en el carácter nulo (`'\0'`). Sin embargo, Java utiliza una aproximación diferente, ya que implementa las cadenas como objetos de tipo **`String`**.

La implementación de las cadenas como objetos le permite a Java proporcionar un conjunto completo de operaciones que facilitan la manipulación de las cadenas. Por ejemplo, hay métodos para comparar dos cadenas, buscar una cadena dentro de otra, concatenar dos cadenas y cambiar mayúsculas y minúsculas dentro de una cadena. Además, los objetos de tipo **`String`** se pueden crear de varias formas, ya que existen diferentes constructores para esta clase.

Sin embargo, hay algo que es un poco inesperado, ya que cuando creamos un objeto de tipo **`String`** estamos creando una cadena que no puede ser modificada, es decir, una vez que se ha creado un objeto **`String`** no se pueden cambiar los caracteres que forman esa cadena, y tampoco su longitud. Esto puede parecer una seria restricción, aunque no es así, ya que podemos realizar sobre las cadenas todas las operaciones habituales. Cada vez que es necesario modificar una cadena existente se crea un nuevo objeto **`String`** que contiene las modificaciones, dejando sin utilidad la cadena inicial. Se utiliza este mecanismo porque las cadenas fijas, sobre las que no se pueden realizar cambios, se implementan de manera más eficiente que las que permiten cambios.

Nota: En aquellos casos en los que se necesite una cadena que pueda ser modificada hay una clase alternativa que acompaña a la clase **`String`,** pero que no se verá en estas prácticas.

La clase **`String`** está definida en el paquete **`java.lang`**, por lo que puede ser utilizada de forma automática en todos los programas. Está declarada como **`final`**, lo que significa que no se pueden crear subclases a partir de esta clase. Esto permite ciertas optimizaciones para incrementar el rendimiento en operaciones habituales con cadenas.

Por último, comentar que cuando se dice que las cadenas que son objetos del tipo **`String`** no se pueden cambiar, lo que realmente significa es que el contenido de una instancia **`String`** no se puede cambiar después de haberla creado. Sin embargo, una variable declarada como referencia a **`String`** se puede cambiar en cualquier momento para que apunte a otro objeto **`String`**.

Constructores de la clase String

La clase **`String`** tiene varios constructores. Para crear una cadena vacía se puede utilizar el constructor por defecto.

```
String s = new String();
```

Este ejemplo creará una instancia de **`String`** sin caracteres en ella.

También es habitual querer crear una cadena con un valor inicial. La clase **`String`** proporciona una serie de constructores para poder hacerlo. Para crear un objeto **`String`** inicializado con una matriz de caracteres se utiliza el siguiente constructor:

```
String(char chars[])
```

El siguiente ejemplo permite inicializar **`s`** con la cadena **`"Telematica"`**.

```
char chars[] = {'T','e','l','e','m','a','t','i','c','a'};

String s = new String(chars)
```

Otra forma alternativa de dar un valor a **`s`** es la siguiente:

```
String s = "Telematica"; /* usa un literal */
```

También se puede especificar un subrango de una matriz de caracteres como inicializador utilizando el siguiente constructor:

```
String(char chars[], int inidiceIni, int numChars)
```

Aquí, ***`inidiceIni`*** indica la posición en la que comienza el subrango y ***`numChars`*** especifica el número de caracteres que se utilizan. El siguiente ejemplo inicializa ***`s`*** con los caracteres **`"lema"`**.

```
char chars[] = {'T','e','l','e','m','a','t','i','c','a'};

String s = new String(chars, 2 , 4 )
```

Nota: El contenido de la matriz se copia cuando se crea el objeto **`String`**. Si se modifica el contenido de la matriz después de haber creado la cadena, el objeto **`String`** no se modifica.

Utilizando el siguiente constructor se puede construir un objeto **`String`** que contenga la misma secuencia de caracteres que otro objeto **`String`**.

```
String(String objStr)
```

Aquí, ***`objStr`*** es un objeto **`String`**. Veamos el siguiente ejemplo:

```
package fp2.poo.practica5;

public class Practica5Ejercicio01 {
    public static void main( String args[] ) {
        char chars[] = {'T','e','l','e','m','a','t','i','c','a'};
        char c[] = {'F','u','n','d','a','m','e','n','t','o','s',' ','d','e',' ',
                    'P','r','o','g','r','a','m','a','c','i','o','n',' ', 'I', 'I' };
        String str          = "Fundamentos de Programacion II";
        String s1           = new String(c);
        String s2           = new String(s1);
        String fundamentos  = "Fundamentos ";
        String de           = "de ";
        int    [] fundament = {70,117,110,100,97,109,101,110,116,111,115};
        String    funString = new String(fundament,0,fundament.length);

        System.out.println(new String(chars,2,4));
        System.out.println(funString + " de Programacion II");
        System.out.println("Fundamentos de Programacion II");
        System.out.println(str);
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(c);
        System.out.println(fundamentos  + de + "Programacion II");
    }
}
```

Practica5Ejercicio01.java

Ejecute el ejemplo anterior mediante la siguiente orden, y observe que en todas las invocaciones a `System.out.println`, contienen la misma información, salvo en el primer caso.

`make –f makeP5 ejecuta01`

Métodos de la clase String

La clase **`String`** tiene una gran diversidad de métodos para el tratamiento de cadenas. Algunos de los más usados son los siguientes:

- **`int length()`**

Devuelve el número de caracteres que contiene la cadena.

- **`char charAt(int `*`posicion`*`)`**

Devuelve el carácter que se encuentra en la posición especificada.

- **`void getChars(int posInicial, int posFinal, char[] bufDestino, int destInicio)`**

Extrae más de un carácter a la vez. **`posInicial`** especifica la posición inicial donde comienza la subcadena. **`posFinal`**, especifica la posición siguiente de donde termina la subcadena.

Así, la subcadena contiene los caracteres comprendidos entre las posiciones **`posInicial`** y **`posFinal`**-**1**. La matriz que recibirá los caracteres está especificada por **`bufDestino`**. La posición dentro de **`bufDestino`** en la que se copiará la subcadena la determina **`destInicio`**. Es necesario tener cuidado para asegurar que la matriz **`bufDestino`** es lo suficientemente grande como para guardar todos los caracteres de la subcadena especificada.

- **`boolean equals(Object str)`**

Compara el objeto **`String`** con la cadena **`str`** dada como parámetro. Devuelve **`true`** si las cadenas contienen los mismos caracteres en el mismo orden y **`false`** en caso contrario.

- **`boolean equalsIgnoreCase(String str)`**

Compara el objeto **`String`** con la cadena **`str`** dada como parámetro. Devuelve **`true`** si las cadenas contienen los mismos caracteres en el mismo orden y **`false`** en caso contrario. Sin tener en cuenta mayúsculas y minúsculas.

- **`int compareTo(String str)`**

Compara el objeto **`String`** con la cadena **`str`** dada como parámetro. El resultado de la comparación se interpreta de la siguiente forma:

Valor	**Significado**
Menor que cero	La cadena es menor que **`str`**.
Mayor que cero	La cadena es mayor que **`str`**.
Cero	Las dos cadenas son iguales.

- **`int indexOf(int ch)`**

Busca la primera ocurrencia del carácter **`ch`** y devuelve la posición en la que se encuentra el carácter o **`-1`** en caso de fallo.

- **`int lastIndexOf(int ch)`**

Busca la última ocurrencia del carácter **`ch`** y devuelve la posición en la que se encuentra el carácter o **`-1`** en caso de fallo.

- **`int indexOf(int ch, int fromIndex)`**
- **`int indexOf(String str)`**
- **`int indexOf(String str, int fromIndex)`**
- **`int lastIndexOf(int ch, int fromIndex)`**
- **`int lastIndexOf(String str)`**
- **`int lastIndexOf(String str, int fromIndex)`**

Estas son otras formas de **`indexOf`** y **`lastIndexOf`** que permiten especificar desde donde se empieza a buscar (**`fromIndex`**) y buscar una subcadena (**`str`**) dentro de una cadena.

- **`String substring(int posInicial)`**
- **`String substring(int posInicial, int posFinal)`**

Permite extraer un fragmento de una cadena. **`posInicial`** especifica la posición donde comienza la subcadena y **`posFinal`** la última posición. La cadena devuelta contiene todos los caracteres que hay desde la posición inicial, pero no incluye la posición final.

- **`String concat(String str)`**

Este método crea un nuevo objeto que contiene una cadena formada por la cadena que llama al método más el contenido de **`str`** añadido al final.

- **`String replace(char original, char sustituto)`**

El método **`replace()`** sustituye en la cadena que llama al método todas las ocurrencias del carácter **`original`** por **`sustituto`**.

- **`String trim()`**

Devuelve una copia de la cadena que llama al método eliminando las secuencias de espacios en blanco que aparezcan tanto al principio como al final de la cadena.

- **`String toLowerCase()`**
- **`String toUpperCase()`**

Ambos métodos devuelven un objeto **`String`** que contiene la cadena equivalente, en mayúsculas o minúsculas, del objeto **`String`** que llamó al método. Los caracteres que no son alfabéticos, como los números no se modifican.

Concatenación de cadenas usando el operador +

En general, Java no permite que los operadores se apliquen a los objetos **`String`**. La única excepción es el operador +. Este operador concatena dos cadenas, produciendo como resultado un objeto **`String`**. Esto permite encadenar una serie de operaciones +.

Por ejemplo, el siguiente fragmento de código concatena tres cadenas.

```
String edad = "9";
String s = "Él tiene  "+ edad +  " años.";
System.out.println(s);
```

Este fragmento de código imprime la cadena **`"Él tiene 9 años."`**

La concatenación de cadenas es muy práctica cuando se crean cadenas muy largas. En lugar de permitir que las cadenas largas líen el código fuente, se pueden dividir en partes más pequeñas y concatenarlas utilizando el operador +. Esto se muestra en el ejemplo siguiente:

```
package fp2.poo.practica5;


public class Practica5Ejercicio02 {

    public static void main ( String args[] ) {

        String s = "Esta podria haber sido una " +
                   "linea muy larga que habria ocupado " +
                   "mas de una linea.";
        System.out.println(s);
    }
}
```

Practica5Ejercicio02.java

Para ejecutar el programa use el siguiente comando:

`make –f makeP5 ejecuta02`

Concatenación con otros tipos de datos en las cadenas

Las cadenas se pueden concatenar con otros tipos de datos. Por ejemplo, consideremos una versión modificada del ejemplo del apartado anterior.

```
int edad = 9;
String s = "El tiene  "+ edad +  " años.";
System.out.println(s);
```

En este caso, **`edad`** es un entero en lugar de otro **`String`**. Sin embargo, la salida generada es la misma. Esto se debe a que el valor **`int`** de **`edad`** automáticamente se

convierte en una cadena dentro de un objeto **`String`**. Después, esta cadena se concatena igual que antes.

El compilador convierte un operando en su cadena equivalente cuando otro operando del operador + es una instancia de **`String`**.

Sin embargo, hay que tener cuidado cuando se mezclan otros tipos de operaciones con las expresiones de concatenación de cadenas ya que se pueden obtener resultados sorprendentes. Consideremos el siguiente fragmento de código:

```
String s = "cuatro: " + 2 + 2;
System.out.println(s);
```

Este fragmento imprime:

```
cuatro: 22
```

en lugar de

```
cuatro: 4
```

que posiblemente era el resultado esperado. Sin embargo, la precedencia de operadores hace que en primer lugar se concatene la cadena "cuatro: " con la cadena equivalente del primer número 2. Después, se concatena este resultado con la cadena equivalente del segundo número 2. Para realizar primero la suma de enteros es necesario utilizar paréntesis.

```
String s = "cuatro: " + (2 + 2);
```

Ahora, **s** contiene la cadena "cuatro: 4".

Ejemplos de programas que usan los métodos de la clase String:

Programa que imprime la longitud de una cadena:

```
package fp2.poo.practica5;

public class Practica5Ejercicio03 {
    public static void main(String[] args) {
        String str = null;
        char chars[] = {'F','u','n','d','a','m','e','n','t','s',' ','d','e',' ',
                'P','r','o','g','r','a','m','a','c','i','o','n', ' ','I','I'};
        try{
            str.length();
            System.out.println("Este codigo nunca se ejecuta");
        } catch (NullPointerException e){
            System.out.println("invocacion incorrecta del metodo length " + e);
        }
        str = new String(chars);
        System.out.println(str.length());
    }
}
```

Practica5Ejercicio03.java

Para ejecutar el programa use el siguiente comando:

`make –f makeP5 ejecuta03`

Recuerde que la invocación a un método mediante una referencia nula (**`null`**) genera una excepción del tipo **`NullPointerException`**.

Programa que compara dos cadenas:

```
package fp2.poo.practica5;

public class Practica5Ejercicio04 {
    public static void main (String args[]){
        String s1 = "Hola";
        String s2 = "Hola";
        String s3 = "Adios";
        String s4 = "HOLA";

        System.out.println(s1+" equals " + s2 + "->" + s1.equals(s2));
        System.out.println(s1+" equals " + s3 + "->" + s1.equals(s3));
        System.out.println(s1+" equals " + s4 + "->" + s1.equals(s4));
        System.out.println(s1+" equalsIgnoreCase " +
                           s4 + "->" + s1.equalsIgnoreCase(s4));
    }
}
```

Practica5Ejercicio04.java

Para ejecutar el programa use el siguiente comando:

`make –f makeP5 ejecuta04`

Programa que sustituye en una cadena todas las ocurrencias de una subcadena por otra cadena dada.

```
package fp2.poo.practica5;

public class Practica5Ejercicio05 {
    public static void main (String args[]){
        String orig = "Sustituye todas las subcadenas que encuentra";
        String busca = "as";
        String sub = "XXX";
        String result= "";
        int i=0;

        System.out.println(orig);

        do {//sustituye todas las subcadenas que encuentra
            i=orig.indexOf(busca);
            if (i !=-1){
                result = orig.substring(0,i);
                result = result + sub;
                result = result + orig.substring(i+busca.length());
                orig = result;
                System.out.println(orig);
            }
        } while (i != -1);
    }
}
```

Practica5Ejercicio05.java

Para ejecutar el programa use el siguiente comando:

`make –f makeP5 ejecuta05`

Interpretar el resultado.

Programa que imprime una cadena original, su equivalente en mayúsculas y su equivalente en minúsculas

```
package fp2.poo.practica5;

public class Practica5Ejercicio06 {
    public static void main (String args[]){
        String s = "Fundamentos de Programacion II.";

        System.out.println("Original: "+s);

        String upper = s.toUpperCase();
        String lower = s.toLowerCase();

        System.out.println("Mayusculas: "+ upper);
        System.out.println("Minusculas: "+ lower);
    }
}
```

Practica5Ejercicio06.java

Para ejecutar el programa use el siguiente comando:

make –f makeP5 ejecuta06

El método equals() de la clase Object y ==

Es importante entender que el método **equals()** y el operador == realizan dos operaciones distintas. Como ya hemos explicado, mientras que el método **equals()** compara los caracteres de un objeto **String**, el operador == compara dos referencias de objeto para ver si hacen referencia a la misma instancia. El siguiente programa muestra como dos objetos **String** diferentes pueden contener los mismos caracteres, pero las referencias a estos objetos no serán iguales.

```
package fp2.poo.practica5;

public class Practica5Ejercicio07 {
    public static void main (String args[]){
        String s1 = "Hola";
        String s2 = new String (s1);
        System.out.println(s1+" equals " + s2 + "->" +
s1.equals(s2));
        System.out.println(s1+" == " + s2 + "->" + (s1==s2));
    }
}
```

Practica5Ejercicio07.java

Para ejecutar el programa use el siguiente comando:

make –f makeP5 ejecuta07

El resultado es el siguiente:

```
Hola equals Hola->true

Hola == Hola->false
```

La variable **s1** hace referencia a una instancia **String** creada a partir de la cadena **"Hola"**. El objeto al que hace referencia **s2** se crea utilizando **s1** como inicializador. Por tanto, el contenido de los dos objetos **String** será idéntico, pero realmente son objetos distintos. Esto significa que **s1** y **s2** no hacen referencia al mismo objeto y, por tanto, el operador == devolverá **false**, como muestra la salida del ejemplo anterior.

La clase **String** dispone de una amplia gama de métodos para la manipulación de las cadenas de caracteres. Para una referencia completa consultar la documentación del API del JDK.

4. ENVOLVENTES DE LOS TIPOS SIMPLES

Java utiliza los tipos simples por razones de rendimiento. Estos tipos de datos no forman parte de la jerarquía de objetos. Se pasan a los métodos por valor y no se pueden pasar directamente por referencia. Además, no es posible que dos objetos se refieran a la misma instancia de un **`int`**.

A veces, será necesario crear una representación como objeto de uno de estos tipos simples. Para almacenar un tipo simple en un objeto se utilizan las clases envolventes (*wrap*). En esencia, estas clases encapsulan, o envuelven, los tipos simples dentro de una clase.

En java se utilizan las siguientes clases envolventes:

- **`Boolean`**
- **`Character`**
- **`Integer`**
- **`Long`**
- **`Double`**
- **`Float`**

En esta práctica no se verán todos los constructores ni todos los métodos correspondientes a estas clases, pero se dan los siguientes ejemplos:

Programa que crea e inicia varios enteros de distintas formas

```
package fp2.poo.practica5;

class Practica5Ejercicio08 {
    public static void main (String args[]) {
        String s            = "22";
        int    j            = 0;
        int    i            = 0;
        int    resultado    = 0;
        /*
         * Usa el constructor con un parametro de tipo String
         */
        Integer miEntero    = new Integer(s);
        Integer miResultado = null;

        System.out.println("el valor del entero "+ miEntero);
        /*
         * Usa el metodo intValue() para obtener el tipo simple
         */
        j = miEntero.intValue();

        /*
         * Usa el metodo estatico de Integer para obtener un entero (int)
         * a partir de un objeto de tipo String.
         */
        i = Integer.parseInt(s);
        resultado = i + j;
        System.out.println("i + j = " + resultado);

        /*
         * Usa el constructor con un parametro de tipo int
         */
        miResultado = new Integer(resultado);
        System.out.println("el valor del resultado es "+ miResultado );
    }
   -  }
```

Practica5Ejercicio08.java

Para ejecutar el programa use el siguiente comando:

make –f makeP5 ejecuta08

Programa que crea e inicia varios reales de precisión doble (`double`) de distintas formas.

```
package fp2.poo.practica5;

public class Practica5Ejercicio09 {
    public static void main ( String args[] ) {
        Double double1  = null;
        Double double2  = null;
        Double double3  = null;
        Double double4  = null;
        Double double5  = null;
        double d1       = 3.3d;
        double d2       = 0.0d;
        /*
         * Constructor con un tipo double
         */
        double1 = new Double(2.2);
        System.out.println("double1 = new Double (2.2):\t\t" + double1);

        /*
         * Constructor con un tipo double
         */
        double2  = new Double(d1);
        System.out.println("double2  = new Double(d1): \t\t" + double2);

        /*
         * Constructor con un tipo double
         */
        double3  = new Double("10.14d");
        System.out.println("double3  = new Double(\"10.14d\"): \t"
                           + double3);

        /*
         * Usa el metodo doubleValue() para obtener el tipo simple
         */
        d1 = double3.doubleValue();
        System.out.println("d1 = double3.doubleValue() :  \t\t" + d1  );

        /*
         * Usa el metodo estatico valueOf() para obtener un
         * objeto del tipo Double a partir de una cadena.
         */
        double4  = Double.valueOf("10.14d");
        System.out.println("double4  = Double.valueOf(\"10.14d\"): \t"
                           + double4  );

        /*
         * Usa el metodo estatico parseDouble() para obtener
         * un objeto del tipo Double
         * a partir de una cadena.
         */
        d2  =Double.parseDouble( "10.14d" );
        System.out.println("d2 = Double.parseDouble(\"10.14d\"):  \t"
                           + d2 );
    }
}
```

Practica5Ejercicio09.java

Para ejecutar el programa use el siguiente comando:

`make –f makeP5 ejecuta09`

Programa que para cada carácter de una matriz indica si es dígito, letra, espacio, mayúscula o minúscula:

```
package fp2.poo.practica5;

public class Practica5Ejercicio10 {
    public static void main ( String args[] ) {
        char a[] = {'F','u','n','d','a','m','e','n','t','s',
                    ' ','d','e',' ',
                    'P','r','o','g','r','a','m','a','c','i','o','n',
                    ' ','I','I'};

        for(int i = 0; i < a.length; i++){
            if(Character.isDigit(a[i]))
                System.out.println(a[i]+" es un digito");
            if(Character.isLetter(a[i]))
                System.out.println(a[i]+" es una letra");
            if(Character.isWhitespace(a[i]))
                System.out.println(a[i]+" es un espacio");
            if(Character.isUpperCase(a[i]))
                System.out.println(a[i]+" es mayuscula");
            if(Character.isLowerCase(a[i]))
                System.out.println(a[i]+" es minuscula");
            System.out.println();
        }
    }
}
```

Practica5Ejercicio10.java

Para ejecutar el programa use el siguiente comando:

`make –f makeP5 ejecuta09`

Ejercicios.

1. Cree un directorio de nombre su uvus y en este directorio cree los directorios **`bin`** y **`jar`**. Descomprima el fichero **`CodigoDeLaParteFinalDeLaPractica5.zip`** proporcionado, para tener el directorio **`src`** en la carpeta de nombre su uvus. Dada la interfaz `SaldoInterfaz.java` del paquete `fp2.poo.utilidades`, cuyo código aparece a continuación:

```
package fp2.poo.utilidades;


public interface SaldoInterfaz{

    /**
     * Descripcion: Devuelve como double el saldo.
     */
    public double getSaldo();

    /**
     * Descripcion: Devuelve como Double el saldo.
     */
    public Double getSaldoDouble();

    /**
     * Descripcion: Devuelve configura con Double el saldo.
     */
    public void setSaldo(Double d);

    /**
     * Descripcion: Devuelve configura con double el saldo.
     */
    public void setSaldo(double d);
}
```

SaldoInterfaz.java

Implemente la clase **`Saldo`** en el paquete **`fp2.poo.pfpooXXX`**, siendo **`XXX`** el **login** del alumno proporcionado por el Centro de Cálculo (CDC).
La clase **`Saldo`** mantiene la información relacionada con el saldo de una cuenta bancaria. Deberá tener un atributo privado para almacenar dicha información.

2. Dada la interfaz `NumeroDeCuentaInterfaz.java` del paquete `fp2.poo.utilidades`, cuyo código aparece a continuación:

```
package fp2.poo.utilidades;

/*
 * En fp2.poo.pfpoofp2.Excepciones.NumeroDeCuentaIncorrectaExcepcion
 * pfpoofp2 debe ser sustituido por pfpooXXX siendo XXX el login
 * del alumno proporcinado por el Centro de Calculo (CDC).
 */
import  fp2.poo.pfpoofp2.Excepciones.NumeroDeCuentaIncorrectaExcepcion;

public interface NumeroDeCuentaInterfaz {

    /**
     * Descripcion: Devuelve el numero de cuenta como String.
     */
    public String getNumeroDeCuenta();

    /**
     * Descripcion: Configura el numero de cuenta.
     */
    public void setNumeroDeCuenta( String numeroDeCuenta ) throws
NumeroDeCuentaIncorrectaExcepcion;
}
```

NumeroDeCuentaInterfaz.java

Implemente la clase **NumeroDeCuenta** en el paquete **fp2.poo.pfpooXXX**, siendo **XXX** el **login** del alumno proporcionado por el Centro de Cálculo (CDC).
La clase **NumeroDeCuenta** mantiene la información relacionada con el número de una cuenta bancaria. El número de una cuenta bancaria consta exactamente de 20 dígitos decimales. Si el número de dígitos utilizados es menor o mayor al configurar el objeto se deberá lanzar la excepción NumeroDeCuentaIncorrectaExcepcion.
Implemente la interfaz NumeroDeCuentaIncorrectaExcepcion, en el subpaquete Excepciones del paquete (fp2.poo.pfpoo**XXX**) siendo XXX el login del alumno.

3. Dada la interfaz DniInterfaz.java del paquete fp2.poo.utilidades, cuyo código aparece a continuación:

```
package fp2.poo.utilidades;

/*
 * En fp2.poo.pfpoofp2.Excepciones.DniIncorrectoExcepcion
 * pfpoofp2 debe ser sustituido por pfpooXXX siendo XXX el login
 * del alumno proporcinado por el Centro de Calculo (CDC).
 */
import fp2.poo.pfpoofp2.Excepciones.DniIncorrectoExcepcion;

/**
 *
 * Descripcion: La interfaz Dni mantiene los metodos
 * para manejar objetos del tipo Dni.
 */
public interface DniInterfaz {

    /**
     * Descripcion: metodo que proporciona un valor
     *              para configurar  el dni.
     */
    public void setDni( String dni ) throws DniIncorrectoExcepcion;

    /**
     * Descripcion: metodo que devuelve como String el dni.
     */
    public String getDni();

}
```

DniInterfaz.java

Implemente la clase **Dni** en el paquete **fp2.poo.pfpooXXX**, siendo **XXX** el **login** del alumno proporcionado por el Centro de Cálculo (CDC).
La clase **Dni** mantiene la información relacionada con una identificación que consta de 8 caracteres, en forma de dígitos decimales, seguido de un carácter alfabético.
El Dni consta exactamente de 8 dígitos decimales y un carácter alfabético. Si el número de dígitos utilizado es menor o mayor al configurar el objeto se deberá lanzar la excepción DniIncorrectoExcepcion.

Implemente la clase `DniIncorrectoExcepcion`, en el subpaquete `Excepciones` del paquete (`fp2.poo.pfpoo`**`XXX`**) siendo XXX el **login** del alumno.

Trabajo a entregar

1. Implemente la clase **`Main`** en el fichero **`Main.java`** del paquete **`fp2.poo.pfpooXXX`**, siendo **`XXX`** el **login** del alumno, para crear en el método **`main`** objetos de la clase **`Saldo`**, **`Dni`** y **`NumeroDeCuenta`**, implementados en la anterior sección (Ejercicios) en los apartados 1, 2 y 3. Proporcione un makefile para realizar la compilación y ejecución.
2. Comprima el directorio de nombre su uvus en el fichero uvus.zip y entréguelo en la actividad correspondiente a la práctica.