

Fundamentos de la Programación II

Programación Orientada a Objetos

Practica 0:
Convenciones al código para el lenguaje de
Programación Java

Índice

1 Introducción

1.1 Presentación del documento

1.2 ¿Por qué tener convenciones de código?

2 Nombres de ficheros

2.1 Extensiones de los ficheros

2.2 Nombres comunes para ficheros

3 Organización de los ficheros

3.1 Ficheros fuente Java

3.1.1 Comentarios de comienzo

3.1.2 Sentencias package e import

3.1.3 Declaraciones de clases e interfaces

4 Indentación

4.1 Longitud de la línea

4.2 Rompiendo líneas

5 Comentarios

5.1 Formatos de los comentarios de implementación

5.1.1 Comentarios de bloque

5.1.2 Comentarios de una línea

5.1.3 Comentarios de remolque

5.1.4 Comentarios de fin de línea

5.2 Comentarios de documentación

6 Declaraciones

6.1 Cantidad por línea

6.2 Inicialización

6.3 Colocación

6.4 Declaraciones de clases e interfaces

7 Sentencias

7.1 Sentencias simples

7.2 Sentencias compuestas

7.3 Sentencias return

7.4 Sentencias if, if-else, if else-if else

7.5 Sentencias for

7.6 Sentencias while

7.7 Sentencias do-while

7.8 Sentencias switch

7.9 Sentencias try-catch

8 - Espacios en blanco

8.1 Líneas en blanco

8.2 Espacios en blanco

9 - Convenios de nombrado

10 - Hábitos de programación

10.1 Proporcionando acceso a variables de instancia y de clase

10.2 Referencias a variables y métodos de clase

10.3 Constantes

10.4 Asignaciones de variables

10.5 Hábitos varios

10.5.1 Paréntesis

10.5.2 Valores de retorno

10.5.3 Expresiones antes de `?' en el operador condicional

10.5.4 Comentarios especiales

11 - Ejemplos de código

11.1 Ejemplo de fichero fuente Java

Convenciones de código para Java

1 - Introducción

1.1 Presentación del documento

El siguiente documento describe los convenios para escribir el código Java que se va tener que seguir, en la asignatura de Fundamentos de la Programación II. Todo código que se escriba debe ajustarse a lo descrito en este documento.

Este documento refleja los estándares de codificación del lenguaje Java presentados en *Java Language Specification*, de Sun Microsystems, Inc. Los mayores contribuidores son Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, y Scott Hommel. Y es el que se va a utilizar como convenio para especificar el código en el lenguaje de programación Java.

1.2 ¿Por qué tener convenciones de código?

Las convenciones de código son importantes para los programadores por un gran número de razones:

- El 80% del coste del código de un programa va a su mantenimiento.
- Casi ningún software lo mantiene toda su vida el autor original.
- Las convenciones de código mejoran la lectura del software, permitiendo a los ingenieros entender nuevo código mucho más rápido y más a fondo.
- Si distribuyes tu código fuente como un producto, necesitas asegurarte de que está bien hecho y presentado como cualquier otro producto.

Para que funcionen las convenciones, TODOS los programadores deben seguir las.

2 - Nombres de ficheros

Esta sección muestra las extensiones y nombres de ficheros más comúnmente usadas.

2.1 Extensiones de los ficheros

El software Java usa las siguientes extensiones para los ficheros:

Tipo de archivo	Extensión
Fuente Java	.java
Bytecode de Java	.class

2.2 Nombres comunes para ficheros

Los nombres de ficheros más utilizados incluyen:

Nombre de fichero	Uso
makefile	El nombre preferido para ficheros "make". Usamos <code>make</code> para construir nuestro software.
README	El nombre preferido para el fichero que resume los contenidos de un directorio particular.

3 - Organización de los ficheros

Un fichero consiste de secciones que deben estar separadas por líneas en blanco y comentarios opcionales que identifican cada sección. Los ficheros de más de 2000 líneas son incómodos y deben ser evitados.

Para ver un ejemplo de un programa de Java debidamente formateado, ver "Ejemplo de fichero fuente Java" en el punto 11 de este documento.

3.1 Ficheros fuente Java

Cada fichero fuente Java contiene una única clase o interfaz pública. Cuando algunas clases o interfaces privadas están asociadas a una clase pública, pueden ponerse en el mismo fichero que la clase pública. La clase o interfaz pública debe ser la primera del fichero.

Los ficheros fuentes Java tienen las siguientes secciones:

- Comentarios de comienzo (ver "Comentarios de comienzo" en el punto 3.1.1).
- Sentencias package e import. (ver “Sentencias package e import” en 3.1.2).
- Declaraciones de clases e interfaces (ver "Declaraciones de clases e interfaces" en el punto 3.1.3).

3.1.1 Comentarios de comienzo

Todos los ficheros fuente deben comenzar con un comentario (al estilo lenguaje C) en el que se muestra el nombre de la clase, información de la versión, fecha, y copyright:

```
/*
 * Nombre de la clase
 *
 * Información de la version
 *
 * Fecha
 *
 * Copyright
 */
```

3.1.2 Sentencias package e import

La primera línea de los ficheros fuente Java que no sea un comentario, es la sentencia `package`. Después de ésta, pueden seguir varias sentencias `import`. Por ejemplo:

```
package java.awt;  
  
import java.awt.peer.CanvasPeer;
```

3.1.3 Declaraciones de clases e interfaces

La siguiente tabla describe las partes de la declaración de una clase o interfaz, en el orden en que deberían aparecer. Ver "Ejemplo de fichero fuente Java" al final de este documento para un ejemplo que incluye comentarios.

Partes de la declaración de una clase o interfaz	Notas
1 Comentario de documentación de la clase o interfaz (<code>/**...*/</code>)	Ver "Comentarios de documentación" en el punto 5.2 para más información sobre lo que debe aparecer en este comentario.
2 Sentencia <code>class</code> o <code>interface</code>	
3 Comentario de implementación de la clase o interfaz si fuera necesario (<code>/*...*/</code>)	Este comentario debe contener cualquier información aplicable a toda la clase o interfaz que no era apropiada para estar en los comentarios de documentación de la clase o interfaz.
4 Variables de clase (<code>static</code>)	Primero las variables de clase <code>public</code> , después las <code>protected</code> , después las de nivel de paquete (sin modificador de acceso), y después las <code>private</code> .
5 Variables de instancia	Primero las <code>public</code> , después las <code>protected</code> , después las de nivel de paquete (sin modificador de acceso), y después las <code>private</code> .
6 Constructores	
7 Métodos	Estos métodos se deben agrupar por funcionalidad más que por visión o accesibilidad. Por ejemplo, un método de clase privado puede estar entre dos métodos públicos de instancia. El objetivo es hacer el código más legible y comprensible.

4 – Indentación

Se deben emplear cuatro espacios como unidad de indentación. La construcción exacta de la indentación (espacios en blanco contra tabuladores) no se especifica. Los tabuladores deben ser exactamente cada 8 espacios (no cada 4).

4.1 Longitud de la línea

Evitar las líneas de más de 80 caracteres, ya que no son manejadas bien por muchas terminales y herramientas.

4.2 Rompiendo líneas

Cuando una expresión no entra en una línea, romperla de acuerdo con estos principios:

- Romper después de una coma.
- Romper antes de un operador.
- Preferir roturas de alto nivel que de bajo nivel.
- Alinear la nueva línea con el comienzo de la expresión al mismo nivel de la línea anterior.
- Si las reglas anteriores llevan a código confuso o a código que se aglutina en el margen derecho, indentar justo 8 espacios en su lugar.

Ejemplos de roturas de alto nivel y bajo nivel:

```
for (int i = 0, int j = 0;
     ((i < DIMENSION_1) && (i < DIMENSION_1)); i++, j++) {
    sentencias;
}                                // PREFERIDA LA ROTURA DE ALTO NIVEL

for (int i = 0, int j = 0; ((i < DIMENSION_1)
     && (i < DIMENSION_1)); i++, j++) {
    sentencias;
}                                // EVITAR ROTURAS DE BAJO NIVEL
```

Ejemplos de como romper la llamada a un método:

```
var = unMetodo1(expresionLarga1,
                  unMetodo2(expresionLarga2,
                            expresionLarga3));

var = unMetodo1(expresionLarga1, unMetodo2(expresionLarga2,
                                              expresionLarga3));
```

Ahora dos ejemplos de ruptura de líneas en expresiones aritméticas. Se prefiere el primero, ya que el salto de línea ocurre fuera de la expresión que encierra los paréntesis.

```
nombreLargo1 = nombreLargo2
    * (nombreLargo3 + nombreLargo4 - nombreLargo5)
    + 4 * nombreLargo6; // PREFERIDA

nombreLargo1 = nombreLargo2 * (nombreLargo3 + nombreLargo4
    - nombreLargo) + 4 * nombreLargo6; // EVITAR
```

Ahora dos ejemplos de indentación en declaraciones de métodos. El primero es el caso convencional. El segundo conduciría la segunda y la tercera línea demasiado hacia la derecha con la indentación convencional, así que en su lugar se usan 8 espacios de indentación.

```
//INDENTACION CONVENCIONAL
unMetodo(int unArg, Object otroArg, String todaviaOtroArg,
          Object yOtroMas) {
    sentencias;
}

//INDENTACION DE 8 ESPACIOS PARA EVITAR GRANDES INDENTACIONES
private static synchronized metodoDeNombreMuyLargo(int unArg,
          Object otroArg, String todaviaOtroArg,
          Object yOtroMas) {
    sentencias;
}
```

La ruptura de líneas para sentencias `if` deberá seguir generalmente la regla de los 8 espacios, ya que la indentación convencional (4 espacios) hace difícil ver el cuerpo. Por ejemplo:

```
//NO USAR ESTA INDENTACION
if ((condicion1 && condicion2)
    || (condicion3 && condicion4)
    || !(condicion5 && condicion6)) { //MALOS SALTOS
    hacerAlgo(); //HACEN ESTA LINEA FACIL DE OLVIDAR
}

//USAR MEJOR ESTA INDENTACION
if ((condicion1 && condicion2)
    || (condicion3 && condicion4)
    || !(condicion5 && condicion6)) {
    hacerAlgo();
}

//O USAR ESTA
if ((condicion1 && condicion2) || (condicion3 && condicion4)
    || !(condicion5 && condicion6)) {
    hacerAlgo();
}
```

Hay tres formas aceptables de formatear expresiones ternarias:

```
alpha = (unaLargaExpresionBooleana) ? beta : gamma;
alpha = (unaLargaExpresionBooleana) ? beta
                                : gamma;
alpha = (unaLargaExpresionBooleana)
        ? beta
        : gamma;
```

5 – Comentarios

Los programas Java pueden tener dos tipos de comentarios:

1. comentarios de implementación y
2. comentarios de documentación.

Los comentarios de implementación son aquellos que también se encuentran en C++, delimitados por `/*...*/`, y `//`. Los comentarios de documentación (conocidos como "doc comments") existen sólo en Java, y se limitan por `/**...*/`. Los comentarios de documentación se pueden exportar a ficheros HTML con la herramienta `javadoc`.

Los comentarios de implementación son para comentar nuestro código o para comentarios acerca de una implementación particular. Los comentarios de documentación son para describir la especificación del código, libre de una perspectiva de implementación, y para ser leídos por desarrolladores que pueden no tener el código fuente a mano.

Se deben usar los comentarios para dar descripciones de código y facilitar información adicional que no es legible en el propio código. Los comentarios deben contener sólo información que es relevante para la lectura y entendimiento del programa. Por ejemplo, información sobre cómo se construye el paquete correspondiente o en qué directorio reside, no debe ser incluida como comentario.

Son apropiadas las discusiones sobre decisiones de diseño no triviales o no obvias, pero hay que evitar duplicar información que está presente (de forma clara) en el código ya que es fácil que los comentarios redundantes se queden desfasados. En general, evitar cualquier comentario que pueda quedar desfasado a medida que el código evoluciona.

Nota: La frecuencia de comentarios a veces refleja una pobre calidad del código. Cuando se sienta obligado a escribir un comentario, considere rescribir el código para hacerlo más claro.

Los comentarios no deben encerrarse en grandes cuadrados dibujados con asteriscos u otros caracteres.

5.1 Formatos de los comentarios de implementación

Los programas pueden tener cuatro estilos de comentarios de implementación: de bloque, de una línea, de remolque, y de fin de línea.

5.1.1 Comentarios de bloque

Los comentarios de bloque se usan para dar descripciones de ficheros, métodos, estructuras de datos y algoritmos. Los comentarios de bloque se podrán usar al comienzo de cada fichero y antes de cada método. También se pueden usar en otros lugares, tales como el interior de los métodos. Los comentarios de bloque en el interior de una función o método deben ser indentados al mismo nivel que el código que describen. Un comentario de bloque debe ir precedido por una línea en blanco que lo separe del resto del código.

```
/*
 * Aquí hay un comentario de bloque.
 */
```

5.1.2 Comentarios de una línea

Pueden aparecer comentarios cortos de una única línea indentados al nivel del código que siguen. Si un comentario no se puede escribir en una única línea, debe seguir el formato de los comentarios de bloque (ver sección 5.1.1). Un comentario de una sola línea debe ir precedido de una línea en blanco. Aquí un ejemplo de comentario de una sola línea en código Java (ver también "Comentarios de documentación" en la sección 5.2):

```
if (condicion) {

    /* Código de la condicion. */
    ...
}
```

5.1.3 Comentarios de remolque

Pueden aparecer comentarios muy pequeños en la misma línea que describen, pero deben ser movidos lo suficientemente lejos para separarlos de las sentencias. Si más de un comentario corto aparece en el mismo trozo de código, deben ser indentados con la misma profundidad.

Aquí un ejemplo de comentario de remolque:

```
if (a == 2) {
    b = TRUE;          /* caso especial */
} else {
    c = esPrimo(a);  /* caso cuando a es impar */
}
```

5.1.4 Comentarios de fin de línea

El delimitador de comentario `//` puede convertir en comentario una línea completa o una parte de una línea. No debe ser usado para hacer comentarios de varias líneas consecutivas; sin embargo, puede usarse en líneas consecutivas para comentar secciones de código. Aquí tenéis ejemplos de los tres estilos:

```
if (foo > 1) {  
  
    // Hacer algo.  
    ...  
}  
else {  
    b = false; // Explicar aquí por qué.  
}  
  
//if (bar > 1) {  
//  
// // Hacer algo.  
// ...  
//}  
//else {  
// return false;  
//}
```

5.2 Comentarios de documentación

Nota: Ver "Ejemplo de fichero fuente Java" para ejemplos de los formatos de comentarios descritos aquí.

Para más detalles, ver "How to Write Doc Comments for Javadoc" que incluye información de las etiquetas de los comentarios de documentación (@return, @param, @see):

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

Para más detalles acerca de los comentarios de documentación y javadoc, visitar el sitio web de javadoc:

<http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>

Los comentarios de documentación describen clases Java, interfaces, constructores, métodos y atributos. Cada comentario de documentación se encierra con los delimitadores de comentarios `/**...*/`, con un comentario por clase, interfaz o miembro (método o atributo). Este comentario debe aparecer justo antes de la declaración:

```

/**
 * La clase Ejemplo ofrece ...
 */
public class Ejemplo { ...

```

Nótese que las clases e interfaces de alto nivel no están indentadas, mientras que sus miembros los están. La primera línea de un comentario de documentación (/**) para clases e interfaces no está indentada, sucesivas líneas tienen cada una un espacio de indentación (para alinear verticalmente los asteriscos). Los miembros, incluidos los constructores, tienen cuatro espacios para la primera línea y 5 para las siguientes.

Si se necesita dar información sobre una clase, interfaz, variable o método que no es apropiada para la documentación, usar un comentario de implementación de bloque (ver sección 5.1.1) o de una línea (ver sección 5.1.2) para comentarlo inmediatamente *después* de la declaración. Por ejemplo, detalles de la implementación de una clase deben ir, en un comentario de implementación de bloque *siguiendo* a la sentencia `class`, no en el comentario de documentación de la clase.

Los comentarios de documentación no deben colocarse en el interior de la definición de un método o constructor, ya que Java asocia los comentarios de documentación con la *primera declaración después* del comentario.

6 - Declaraciones

6.1 Cantidad por línea

Se recomienda una declaración por línea, ya que facilita los comentarios. En otras palabras, se prefiere

```

int nivel; // nivel de indentación
int tam;   // tamaño de la tabla

```

antes que

```
int nivel, tam;
```

No poner diferentes tipos en la misma línea. Ejemplo:

```
int foo, fooarray[]; //EVITAR
```

Nota: Los ejemplos anteriores usan un espacio entre el tipo y el identificador. Una alternativa aceptable es usar tabuladores, por ejemplo:

```

int      nivel;          // nivel de indentación
int      tam;            // tamaño de la tabla
Object   entradaActual; // entrada de la tabla seleccionada

```

6.2 Inicialización

Intentar inicializar las variables locales donde se declaran. La única razón para no inicializar una variable donde se declara es si el valor inicial depende de algunos cálculos previos.

6.3 Colocación

Poner las declaraciones solo al principio de los bloques (un bloque es cualquier código encerrado por llaves "{" y "}"). No esperar al primer uso para declararlas; puede confundir a programadores y limitar la portabilidad del código dentro de su visibilidad.

```
void miMetodo() {  
    int int1 = 0; // comienzo del bloque del método  
    if (condicion) {  
        int int2 = 0; // comienzo del bloque del "if"  
        ...  
    }  
}
```

La excepción de la regla son los índices de bucles `for`, que en Java se pueden declarar en la sentencia `for`:

```
for (int i = 0; i < maximoVueltas; i++) { ... }
```

Evitar las declaraciones locales que ocultan declaraciones de niveles superiores. por ejemplo, no declarar la misma variable en un bloque interno:

```
int cuenta;  
...  
miMetodo() {  
    if (condicion) {  
        int cuenta = 0; // EVITAR!  
        ...  
    }  
    ...  
}
```

6.4 Declaraciones de clases e interfaces

Al codificar clases e interfaces de Java, se siguen las siguientes reglas de formato:

- Ningún espacio en blanco entre el nombre de un método y el paréntesis "(" que abre su lista de parámetros.
- La llave de apertura "{" aparece al final de la misma línea de la sentencia de declaración.

- La llave de cierre "}" empieza una nueva línea indentada para ajustarse a su sentencia de apertura correspondiente, excepto cuando no existen sentencias entre ambas, que debe aparecer inmediatamente después de la de apertura "{".

```
class Ejemplo extends Object {
    int ivar1;
    int ivar2;

    Ejemplo(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int metodoVacio() {}
    ...
}
```

- Los métodos se separan con una línea en blanco.

7 – Sentencias

7.1 Sentencias simples

Cada línea debe contener como máximo una sentencia. Ejemplo:

```
argv++;           // Correcto
argc--;           // Correcto
argv++; argc--; // EVITAR!
```

7.2 Sentencias compuestas

Las sentencias compuestas son sentencias que contienen listas de sentencias encerradas entre llaves "{ sentencias }". Ver la siguientes secciones para ejemplos.

- Las sentencias encerradas deben indentarse un nivel más que la sentencia compuesta.
- La llave de apertura se debe poner al final de la línea que comienza la sentencia compuesta; la llave de cierre debe empezar una nueva línea y ser indentada al mismo nivel que el comienzo de la sentencia compuesta.
- Las llaves se usan en todas las sentencias, incluso las simples, cuando forman parte de una estructura de control, como en las sentencias `if-else` o `for`. Esto hace más sencillo añadir sentencias sin incluir errores accidentales por olvidar las llaves.

7.3 Sentencias return

Una sentencia `return` con un valor no debe usar paréntesis a menos que hagan el valor de retorno más obvio de alguna manera. Ejemplo:

```

        return;

        return miDiscoDuro.size();

        return (tamanyo ? tamanyo : tamanyoPorDefecto);
    }
}

```

7.4 Sentencias if, if-else, if else-if else

La clase de sentencias `if-else` debe tener la siguiente forma:

```

if (condicion) {
    sentencias;
}

if (condicion) {
    sentencias;
} else {
    sentencias;
}

if (condicion) {
    sentencia;
} else if (condicion) {
    sentencia;
} else{
    sentencia;
}

```

Nota: Las sentencias `if` usan siempre llaves `{}`. Evitar la siguiente forma, propensa a errores:

```

if (condicion) //EVITAR! ESTO OMITE LAS LLAVES {}
    sentencia;

```

7.5 Sentencias for

Una sentencia `for` debe tener la siguiente forma:

```

for (inicializacion; condicion; actualizacion) {
    sentencias;
}

```

Una sentencia `for` vacía (una en la que todo el trabajo se hace en la inicialización, condición, y actualización) debe tener la siguiente forma:

```

for (inicializacion; condicion; actualizacion);

```

Al usar el operador coma en la inicialización o actualización de una sentencia `for`, evitar la complejidad de usar más de tres variables. Si se necesita, usar sentencias separadas antes de bucle `for` (para la inicialización) o al final del bucle (para la actualización).

7.6 Sentencias while

Una sentencia `while` debe tener la siguiente forma:

```
while (condicion) {  
    sentencias;  
}
```

Una sentencia `while` vacía debe tener la siguiente forma:

```
while (condicion);
```

7.7 Sentencias do-while

Una sentencia `do-while` debe tener la siguiente forma:

```
do {  
    sentencias;  
} while (condicion);
```

7.8 Sentencias switch

Una sentencia `switch` debe tener la siguiente forma:

```
switch (condicion) {  
    case ABC:  
        sentencias;  
        /* este caso se propaga */  
    case DEF:  
        sentencias;  
        break;  
    case XYZ:  
        sentencias;  
        break;  
    default:  
        sentencias;  
        break;  
}
```

Cada vez que un caso se propaga (no incluye la sentencia `break`), añadir un comentario donde la sentencia `break` se encontraría normalmente. Esto se muestra en el ejemplo anterior con el comentario `/* este caso se propaga */`.

Cada sentencia `switch` debe incluir un caso por defecto. El `break` en el caso por defecto es redundante, pero prevé que se propague por error si luego se añade otro caso.

7.9 Sentencias try-catch

Una sentencia `try-catch` debe tener la siguiente forma:

```
try {  
    sentencias;  
} catch (ExceptionClass e) {  
    sentencias;  
}
```

Una sentencia `try-catch` puede ir seguida de un `finally`, cuya ejecución se ejecutará independientemente de que el bloque `try` se halla completado con éxito o no.

```
try {  
    sentencias;  
} catch (ExceptionClass e) {  
    sentencias;  
} finally {  
    sentencias;  
}
```

8 - Espacios en blanco

8.1 Líneas en blanco

Las líneas en blanco mejoran la facilidad de lectura separando secciones de código que están lógicamente relacionadas.

Se deben usar siempre dos líneas en blanco en las siguientes circunstancias:

- Entre las secciones de un fichero fuente
- Entre las definiciones de clases e interfaces.

Se debe usar siempre una línea en blanco en las siguientes circunstancias:

- Entre métodos.
- Entre las variables locales de un método y su primera sentencia.
- Antes de un comentario de bloque (ver sección 5.1.1) o de un comentario de una línea (ver sección 5.1.2).
- Entre las distintas secciones lógicas de un método para facilitar la lectura.

8.2 Espacios en blanco

Se deben usar espacios en blanco en las siguientes circunstancias:

- Una palabra reservada del lenguaje seguida por un paréntesis debe separarse por un espacio. Ejemplo:

```
while (true) {  
    ...  
}
```

Notemos que no se debe usar un espacio en blanco entre el nombre de un método y su paréntesis de apertura. Esto ayuda a distinguir palabras reservadas de llamadas a métodos.

- Debe aparecer un espacio en blanco después de cada coma en las listas de argumentos.
- Todos los operadores binarios excepto el punto “.” deben ir separados de sus operandos mediante espacios. Los operadores unarios como el “menos unario”, el incremento (“++”) y el decremento (“--”) no deben ir nunca separados mediante espacios en blanco de sus operandos. Ejemplo:

```
a += c + d;  
a = (a + b) / (c * d);  
while (d++ == s++) {  
    n++;  
}  
prints("el tamaño es " + foo + "\n");
```

- Las expresiones en una sentencia `for` se deben separar con espacios en blanco. Ejemplo:

```
for (expr1; expr2; expr3)
```

- Los “cast” deben ir seguidos de un espacio en blanco. Ejemplos:

```
miMetodo((byte) unNumero, (Object) x);  
miMetodo((int) (cp + 5), ((int) (i + 3))  
        + 1);
```

9 - Convenios de nombrado

Los convenios de nombrado hacen los programas más entendibles facilitando su lectura. También pueden dar información sobre la función de un identificador, por ejemplo, cuando es una constante, un paquete, o una clase, que puede ser útil para entender el código.

Tipo de Identificador	Reglas de nombrado	Ejemplos
Paquetes	<p>El prefijo del nombre de un paquete se escribe siempre con letras ASCII en minúsculas, y debe ser uno de los nombres de dominio de alto nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que identifican cada país como se especifica en el ISO Standard 3166, 1981.</p> <p>Los sucesivos componentes del nombre del paquete variarán de acuerdo a las convenciones de nombres internas de cada organización. Dichas convenciones pueden especificar que algunos nombres de los directorios correspondan a divisiones, departamentos, proyectos, máquinas o nombres de usuarios.</p>	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
Clases	Los nombres de las clases deben ser sustantivos, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivos. Usar palabras completas, evitar acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo, como URL o HTML).	class Cliente; class ImagenAnimada;
Interfaces	Los nombres de las interfaces siguen la misma regla que las clases.	interface ClienteDelegado; interface Almacen;
Métodos	Los métodos deben ser verbos, cuando son compuestos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras que lo forman en mayúscula.	ejecutar(); ejecutarRapido(); obtenerFondo();

Tipo de Identificador	Reglas de nombrado	Ejemplos
Variables	<p>Excepto las variables, todas las instancias, clases y constantes de clase empezarán con minúscula. Las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas. Los nombres de variables no deben empezar con los caracteres guión bajo "_" o signo del dólar "\$", aunque ambos están permitidos por el lenguaje.</p> <p>Los nombres de las variables deben ser cortos pero con significado. La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador casual su función. Los nombres de variables de un solo carácter se deben evitar, excepto para variables índices temporales. Nombres comunes para variables temporales son <code>i</code>, <code>j</code>, <code>k</code>, <code>m</code> y <code>n</code> para enteros; <code>c</code>, <code>d</code>, y <code>e</code> para caracteres.</p>	<pre>int i; char c; float miAnchura;</pre>
Constantes	<p>Los nombres de las variables declaradas como constantes de clase deben ir totalmente en mayúsculas separando las palabras con un guión bajo ("_").</p>	<pre>static final int MIN = 1; static final int MAX = 9; static final int NUM = 5;</pre>

10 - Hábitos de programación

10.1 Proporcionando acceso a variables de instancia y de clase

No hacer ninguna variable de instancia o clase pública sin una buena razón. A menudo las variables de instancia no necesitan ser asignadas / consultadas explícitamente, esto sucede como efecto lateral de llamadas a métodos.

10.2 Referencias a variables y métodos de clase

Evitar usar un objeto para acceder a una variable o método de clase (static). Usar el nombre de la clase en su lugar. Por ejemplo:

```
metodoDeClase(); //OK
```

```
UnaClase.metodoDeClase();      //OK
unObjeto.metodoDeClase();      //EVITAR!
```

10.3 Constantes

Las constantes numéricas (literales) no deberían ser codificadas directamente, excepto -1, 0, y 1, que pueden aparecer en un bucle `for` como contadores.

10.4 Asignaciones de variables

Evitar asignar el mismo valor a varias variables en la misma sentencia, dificulta su lectura.
Ejemplo:

```
fooBar.fChar = barFoo.Ichar = 'c'; // EVITAR!
```

No usar el operador de asignación en un lugar donde se pueda confundir con el de igualdad.
Ejemplo:

```
if (c++ = d++) { // EVITAR! (Java no lo permite)
    ...
}
```

se debe escribir:

```
if ((c++ = d++) != 0) {
    ...
}
```

No usar asignaciones incrustadas como un intento de mejorar el rendimiento en tiempo de ejecución. Ése es el trabajo del compilador. Ejemplo:

```
d = (a = b + c) + r; // EVITAR!
```

se debe escribir:

```
a = b + c;
d = a + r;
```

10.5 Hábitos varios

10.5.1 Paréntesis

En general es una buena idea usar paréntesis en expresiones que implican distintos operadores para evitar problemas con el orden de precedencia de los operadores. Incluso si parece claro el orden de precedencia de los operadores, podría no ser así para otros. No se debe asumir que otros programadores conozcan el orden de precedencia.

```
if (a == b && c == d) // EVITAR!
```

```
if ((a == b) && (c == d)) // CORRECTO
```

10.5.2 Valores de retorno

Intentar hacer que la estructura del programa se ajuste a su intención. Ejemplo:

```
if (expresionBooleana) {  
    resultado = true;  
} else {  
    resultado =false;  
}  
return resultado;
```

en su lugar se debe escribir

```
return expressionBooleana;
```

Similarmente,

```
if (condicion) {  
    return x;  
}  
return y;
```

se debe escribir:

```
return (condicion ? x : y);
```

10.5.3 Expresiones antes de `?' en el operador condicional

Si una expresión contiene un operador binario antes de “?” en el operador ternario “?:”, se debe colocar entre paréntesis. Ejemplo:

```
(x >= 0) ? x : -x;
```

10.5.4 Comentarios especiales

Usar “xxx” en un comentario para indicar que algo tiene algún error pero funciona. Usar “FIXME” para indicar que algo tiene algún error y no funciona.

11 - Ejemplos de código

11.1 Ejemplo de fichero fuente Java

El siguiente ejemplo muestra como formatear un fichero fuente Java que contiene una única clase pública. Las interfaces se formatean similarmente. Para más información, ver la sección "Declaraciones de clases e interfaces" y "Comentarios de documentación".

```
/*
 * @(#)Bla.java 1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * Más información y descripción del Copyright.
 *
 */

package java.bla;

import java.bla.blabla.BlaBla;

/**
 * La descripción de la clase viene aquí.
 *
 * @version datos de la versión (numero y fecha)
 * @author Nombre Apellido
 */
public class Bla extends OtraClase {
    /* Un comentario de implementación de la clase viene aquí.*/

    /** El comentario de documentación de claseVar1 */
    public static int claseVar1;

    /**
     * El comentario de documentación de classVar2
     * ocupa más de una línea
     */
    private static Object claseVar2;

    /** Comentario de documentación de instanciaVar1 */
    public Object instanciaVar1;

    /** Comentario de documentación de instanciaVar2 */
    protected int instanciaVar2;

    /** Comentario de documentación de instanciaVar3 */
    private Object[] instanciaVar3;

    /**
     * ...Comentario de documentación del constructor Bla...
     */
```

```
 */
public Bla() {
    // ...aquí viene la implementación...
}

/**
 * ...Comentario de documentación del método hacerAlgo...
 */
public void hacerAlgo() {
    // ...aquí viene la implementación...
}

/**
 * ...Comentario de documentación de hacerOtraCosa...
 * @param unParametro descripción
 */
public void hacerOtraCosa(Object unParametro) {
    // ...aquí viene la implementación...
}
}
```



ENTORNO DE PROGRAMACIÓN (I)

OBJETIVO

El objetivo de esta práctica es proporcionar herramientas básicas que permitan desarrollar programas dentro del lenguaje de programación Java. En esta práctica se presentan los comandos más básicos, así como sus opciones más importantes, que se deben conocer y manejar para la correcta realización de las prácticas sobre Java de la asignatura Fundamentos de Programación II.

1. Introducción

Al igual que en las prácticas sobre C realizadas anteriormente en la asignatura Fundamentos de Programación I, estas prácticas se realizarán en los equipos disponibles en el Centro de Cálculo de la Escuela Técnica Superior de Ingeniería y se utilizará Linux como sistema operativo. Por ello se recomienda que, para resolver cualquier duda relativa al manejo del entorno, así como del sistema operativo, se utilice la documentación relativa a las prácticas de la citada asignatura (Apuntes de “*Fundamentos de Programación I*”, Departamento de Ingeniería Telemática, Universidad de Sevilla).

1.1. Entorno de ejecución

Al conjunto de todos los elementos necesarios para ejecutar una aplicación Java se denomina JRE (*Java Standard Edition Runtime Environment*). Este entorno está formado por:

- Máquina virtual de Java (JVM, *Java Virtual Machine*):

Núcleo básico que contiene el intérprete para el lenguaje Java (java). Además, gracias a la máquina virtual de Java, es posible ejecutar un mismo programa en diferentes máquinas con sistemas operativos diferentes.

- Un conjunto de librerías:

Implementan las clases e interfaces básicas y fundamentales definidas en Java, por ejemplo los paquetes `java.lang` y `java.io`.

- Otros componentes.

1.2. Entorno de desarrollo

Al conjunto de todos los elementos necesarios para desarrollar y ejecutar una aplicación Java se denomina JDK (*Java Standard Edition Development Kit*). Este entorno está formado por:

- Entorno de ejecución de Java (*JRE*):
Nos permitirá ejecutar una aplicación Java mediante la ejecución del intérprete (`java`).
- Herramientas para la creación y desarrollo de aplicaciones Java:
Compilador (`javac`), depurador (`jdb`), empaquetador (`jar`) o generador de documentación (`javadoc`) son algunas de estas herramientas. Todas ellas tienen en común que su ejecución se realiza mediante la línea de comandos (ninguna de ellas ofrece una interfaz gráfica).

1.3. Proceso de creación de una aplicación en Java

La creación de una aplicación en Java lleva varios pasos. El primer paso es la fase de análisis del problema. Una vez realizadas las fases de análisis (comprensión del problema) y diseño del programa (que incluye la estructura y algoritmos utilizados), ya se puede acometer la codificación del programa en Java. Una vez que tenemos los ficheros de código fuente (ficheros `.java`), se utiliza el compilador (`javac`) para obtener los ficheros bytecode (ficheros `.class`). Cuando se han obtenido todos los ficheros `.class` a partir de los ficheros `.java`, recurrimos al intérprete (`java`) de la máquina virtual de Java para la ejecución de la aplicación que hemos creado. Este proceso se muestra en la Figura 1.

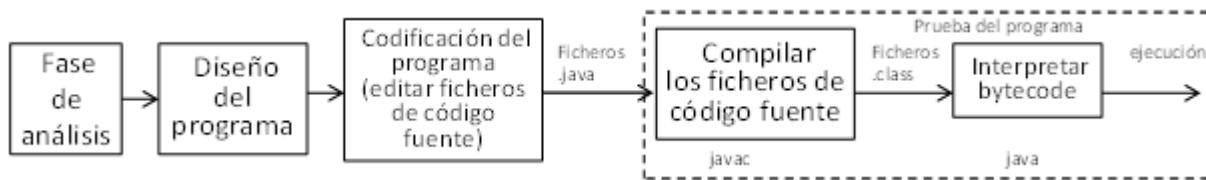


Figura 1: Proceso de creación de una aplicación Java

Por último, ya sólo queda abordar la fase de prueba del programa, con el fin de detectar los errores de codificación cometidos y realizar las modificaciones oportunas.

1.4. Documentación

Existe múltiple bibliografía, tanto en formato físico como formato digital, que proporciona abundante información sobre Java. Oracle ofrece su propia documentación sobre el lenguaje Java, que se recoge en las siguientes páginas web:

- Página principal:
<http://www.oracle.com/technetwork/java/index.html>.
- Documentación general de Java SE (Standard Edition):
<https://www.oracle.com/technetwork/java/javase/overview/index.html>
<http://download.oracle.com/javase/8/docs/index.html>.
- API (*Application Programming Interface*) de Java SE:
<https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

2. Herramientas

Aunque el entorno de desarrollo de Java nos proporciona diversas herramientas, en el ámbito de la asignatura sólo será necesario conocer las herramientas que se detallan a continuación. Se recomienda analizar y comprender en profundidad la finalidad de cada herramienta, así como las opciones más importantes que ofrece, de forma que su futuro uso no suponga una dificultad añadida en el desarrollo de un programa.

2.1. Compilador (javac)

El compilador **javac** se encarga de generar ficheros *bytecode* (ficheros `.class`) a partir de los ficheros de código fuente. La sintaxis de este programa es la siguiente:

```
javac [opciones] ficheros_java
```

donde:

- **[opciones]** modifican el comportamiento del compilador. Algunas de las opciones más frecuentes son las siguientes:
 - d directorio Opción estándar encargada de indicar al compilador el directorio donde ubicar los archivos `.class` tras la compilación de los ficheros fuente `.java`.
 - g Opción estándar encargada de generar toda la información necesaria para depurar el programa mediante el uso del depurador.
 - classpath ruta Opción estándar encargada de indicar al compilador la ruta donde se encuentran los ficheros `.class` necesarios para compilar el fichero de código fuente indicado.
 - v Opción estándar encargada de imprimir la versión del compilador que se está utilizando.
 - Xlint Opción no estándar encargada de imprimir todos los avisos que se produzcan al compilar el código fuente.
- **ficheros_java** indica los ficheros de código fuente que se desean compilar. Estos ficheros deben tener extensión `.java`. De esta forma, la clase ClaseA se debería guardar en un fichero con nombre `ClaseA.java` para que al compilarlo obtengamos el fichero `ClaseA.class`.

Ejemplo de uso

Descargue el fichero **practical1.zip** y descomprímalos. Este fichero contiene archivos para la realización de la primera parte de esta práctica. A continuación, se muestra el fichero de código fuente Ejercicio00.java.

```
public class Ejercicio00 {  
    public static void main( String args[] ) {  
        System.out.println( "El programa se ha ejecutado." );  
    }  
}
```

La orden que se debe ejecutar para realizar la compilación del fichero Ejercicio00.java, indicándole al compilador que nos muestre todos los avisos y que genere toda la información necesaria para poder utilizar el depurador, es la siguiente:

```
javac -g -Xlint ./Ejercicio00.java
```

Observe que el correspondiente fichero .class, en este caso, se genera cuando se realiza la compilación en el directorio de trabajo.

En el caso anterior la compilación se ha realizado encapsulando la clase en un paquete por defecto y que no tiene nombre. Se puede modificar el lugar donde se ubica el fichero fuente. Para ello se ubica el fichero Ejercicio00.java en el directorio fp2/poo/practical en el directorio de inicio de sesión. Esta estructura de directorios va asociado a la estructura de paquetes correspondiente. Se deberá, por tanto, añadir la línea package fp2.poo.practical; para la creación de un paquete con el nombre fp2.poo.practical que contendrá la clase Ejercicio00. Esto se muestra en el código siguiente:

```
/*  
 *  @(#)Ejercicio00.java  
 *  
 *  Fundamentos de Programacion II. GIT.  
 *  Departamento de Ingenieria Telematica  
 *  Universidad de Sevilla  
 *  
 *  Sin copyright  
 */  
  
package fp2.poo.practical;  
  
import java.lang.*;  
  
/**  
 * Descripcion: Esta es una clase de ejemplo.  
 *  
 * @version version 1.0 Abril 2011  
 * @author Fundamentos de Programacion II  
 */  
public class Ejercicio00 {  
    public static void main( String args[] ) {  
        System.out.println( "El programa se ha ejecutado." );  
    }  
}
```

La orden que se debe ejecutar para realizar la compilación del fichero `Ejercicio00.java`, indicándole al compilador que nos muestre todos los avisos y que genere toda la información necesaria para poder utilizar el depurador, es la siguiente:

```
javac -g -Xlint ./fp2/poo/practical/Ejercicio00.java
```

Observe que el correspondiente fichero `.class`, en este caso, se genera cuando se realiza la compilación en el directorio `./fp2/poo/practical/`.

2.2. Intérprete (java)

El intérprete `java` se encarga de ejecutar una aplicación Java. Para ello, inicia todo el entorno necesario para la ejecución, carga la clase indicada y comienza la ejecución del método `main` que debe estar presente en esta clase. La sintaxis de este programa es la siguiente:

```
java [opciones] clase [argumentos]  
java [opciones] -jar fichero_jar [argumentos]
```

donde:

- `[opciones]` modifican el comportamiento del compilador. Algunas de las opciones más frecuentes son las siguientes:

`-classpath ruta` Opción estándar encargada de indicar al intérprete la ruta a los archivos `.class` o `.jar` que se desean ejecutar. El contenido de esta opción sobreescribe el contenido de la variable de entorno `CLASSPATH`. En caso de que no se haya especificado la mencionada variable y tampoco se utilice esta opción, la ruta utilizada será el directorio de trabajo (`.`). Cada uno de los caminos se indicarán separados por `:` (dos puntos). Esta opción también se denomina `-cp ruta`.

`-showversion` Opción estándar encargada de imprimir, previamente a la ejecución de la aplicación, la versión del intérprete que se está utilizando.

`-verbose` Opción estándar que indica al intérprete que muestre de manera explícita la información de cada clase cargada y cada evento del recolector de basura.

`-version` Opción estándar encargada de imprimir la versión del intérprete que se está utilizando.

- `clase` indica al intérprete el nombre de la clase en la que se desea empezar la ejecución.
- `-jar fichero_jar` indica al intérprete el fichero `.jar` que contiene la aplicación Java a ejecutar.
- `[argumentos]` indica los argumentos que se desean pasar a la aplicación por línea de comandos.

Ejemplo de uso

A partir del ejemplo anterior la orden que se debe ejecutar para iniciar la ejecución de la aplicación es la siguiente (esta orden de ejecución se realiza desde el directorio que está por encima de fp2):

```
java fp2.poo.practical.Ejercicio00  
El programa se ha ejecutado.
```

2.3. Empaquetador (jar)

El programa **jar** (*Java ARchive*) se encarga de empaquetar un conjunto de archivos dentro de un único archivo jar, utilizando el formato de fichero ZIP. La sintaxis de este programa depende de la acción que deseamos realizar:

```
jar c[v]f[e] archivo_jar [clase_inicial] [-C directorio] f_class  
                                (Crear un archivo jar)  
  
jar u[v]f[e] archivo_jar [clase_inicial] [-C directorio] f_class  
                                (Actualizar un archivo jar)  
  
jar x[v]f archivo_jar  
                                (Extraer los archivos de un archivo jar)  
  
jar t[v]f archivo_jar  
                                (Ver el contenido de un archivo jar)
```

donde:

- La opción v indica al empaquetador que muestre por pantalla toda la información relativa a la ejecución de la orden del usuario.
- La opción f indica al empaquetador el fichero .jar que se desea crear (con la opción c), actualizar (con la opción u), extraer (con la opción x) o ver el contenido (con la opción t).
- La opción e indica al empaquetador cuál es la clase que contiene la función que inicia el funcionamiento del programa (método main).
- archivo_jar indica el nombre del fichero .jar que se desea crear. Si es necesario, este parámetro contendrá la ruta a la ubicación en la que se debe crear el archivo.
- clase_inicial indica al empaquetador el nombre de la clase que contiene la función main por la que se debe empezar la ejecución de la aplicación.
- La opción -C directorio indica al empaquetador el directorio donde se encuentran los ficheros que se indican a continuación en la orden que se está ejecutando. Este opción podrá aparecer, por tanto, delante del nombre de cada uno de los ficheros a empaquetar.
- f_class indican al empaquetador los ficheros a incluir en el archivo .jar. Se puede indicar tanto los ficheros como los directorios completos (que contendrán archivos .class) que se desean incluir.

Ejemplo de uso

A partir del ejemplo anterior, la orden que se debe ejecutar para empaquetar en un archivo Ejercicio00.jar la aplicación creada es la siguiente:

```
jar cvfe Ejercicio00.jar fp2.poo.practical.Ejercicio00 ./fp2/poo/practical/Ejercicio00.class  
manifest agregado  
agregando: fp2/poo/practical/Ejercicio00.class (entrada = 589) (salida = 367) (desinflado 37%)
```

Una vez creado el archivo Ejercicio00.jar, la orden necesaria para la ejecución de la aplicación directamente a través de este archivo es la siguiente:

```
java -jar Ejercicio00.jar  
El programa se ha ejecutado .
```

2.4. Gestión de documentación (javadoc)

El programa javadoc se encarga de generar de manera automática la documentación correspondiente al código fuente que se ha creado. Para ello el programador debe añadir ciertos comentarios en su código fuente (siguiendo unas sencillas reglas) de forma que el programa javadoc pueda reconocerlas y así crear la documentación correspondiente. Se ampliará en una práctica posterior.

2.5. Depurador (jdb)

El programa jdb es una herramienta muy útil para la depuración de aplicaciones escritas en Java. Se ampliará en una práctica posterior.

3. Entorno

Aunque generalmente no es necesario configurar ningún parámetro del entorno de desarrollo, sí es conveniente conocer determinados aspectos del entorno. Estos aspectos son la variable de entorno CLASSPATH y la estructura de directorios en la que se almacenan todos los elementos que componen el entorno.

3.1. Ruta de ejecución (classpath)

Ya se ha comentado anteriormente la posibilidad de indicar a las herramientas del entorno (java, javac...) la ubicación de los ficheros con los que se desea trabajar. Hay otra posibilidad, aunque no se va a utilizar en estas prácticas: la variable de entorno CLASSPATH. A continuación, se va a indicar cómo trabajar con esta variable de entorno (esto se podía realizar en las versiones previas del lenguaje):

- Consultar el valor almacenado en la variable de entorno CLASSPATH:
El comando con el que se consulta el valor almacenado en la variable de entorno es:

```
echo $CLASSPATH
```

En el resultado que se obtiene es donde se encuentran todos los directorios, separados por el carácter :, que dan valor a la variable de entorno. En la configuración del CdC esa variable de entorno no devuelve ningún valor.

- Establecer la variable de entorno CLASSPATH:

El comando con el que se da contenido a la variable de entorno es:

```
export CLASSPATH=directorio:directorio:...
```

representa al directorio que se desea añadir como valor a la variable de entorno.

- Eliminar el contenido de la variable de entorno CLASSPATH:

El comando con el que se elimina el contenido de la variable de entorno es:

```
unset CLASSPATH
```

Por último, es necesario indicar que el tiempo de vida de la variable de entorno es el terminal en el que se ha fijado. Por tanto, cada vez que se abra un terminal es necesario fijar de nuevo el valor que queramos dar a la variable de entorno CLASSPATH.

3.2. Estructura de archivos del JDK

Con el objetivo de proporcionar una visión más completa del entorno de programación JDK, a continuación, se muestran los directorios más importantes que forman parte del mismo:

/usr/lib/jvm/java-11-openjdk-amd64/

Directorio donde se almacena todo el software que compone el entorno.

/usr/lib/jvm/java-11-openjdk-amd64/bin/

Directorio donde se almacenan todos los ejecutables del entorno.

/usr/lib/jvm/java-11-openjdk-amd64/lib/

Directorio donde se almacenan las clases del entorno que no pertenecen al núcleo de la máquina virtual.

4. Ejercicios

4.1. Uso de las herramientas básicas

1. Use los ficheros proporcionados en **practica1.zip** para esta práctica (**Calculadora.java** y **Ejercicio01.java**) cuyo contenido se muestra a continuación.

```
/*
 *  @( #) Ejercicio01.java
 *
 *  Fundamentos de Programacion II. GIT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *
 */

package fp2.poo.practical;

import java.lang.*;


/**
 * Descripcion: Esta es una clase de prueba que contiene el metodo
 *               main para probar la clase Calculadora.
 *
 * @version version 1.0 Abril 2011
 * @author Fundamentos de Programacion II
 */
public class Ejercicio01 {

    /**
     * Este metodo invoca a la clase Calculadora.
     * Realiza la instanciación de la clase e invoca
     * sus metodos.
     */
    public static void main( String args[] ) {
        int i = 0; /* Variables locales a main */
        int j = 0;
        Calculadora calc = new Calculadora ();

        /* Primera operacion : 1 + 2 ( Resultado : 3 ).*/
        i = 1;
        j = 2;
        System.out.println ("El resultado de sumar " + i + " y "
                           + j + " es " + calc.suma( i, j));

        /* Segunda operacion : Factorial del resultado anterior .
         * ( Resultado : 3! = 6)
         */
        i = calc.getMemoria( );
        j = calc.factorial( i );
        System.out.println( "El factorial de " + i + " es " + j);

        /* Tercera operacion : Dividimos el ultimo resultado por 4.
         * ( Resultado : 1).
         */
        System.out.println( "El resultado de dividir " + j +
                           " y 4 es " + calc.divide( j, 4));

        /* Cuarta operacion : Sumamos el contenido de la memoria a 20.
         * ( Resultado : 21).
         */
        i = 20;
        System.out.println( "El resultado de sumar "
                           + calc.getMemoria() + " y " + i
                           + " es " + calc.suma( i ));
    }
}
```

```

/*
 *  @(#)Calculadora.java
 *
 *  Fundamentos de Programacion II. GIT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *
 */

package fp2.poo.practical;

import java.lang.*;


/**
 * Descripcion: Esta una clase es un ejemplo de implementacion y uso
 *               de los metodos de una clase.
 *
 *
 * @version version 1.0 Abril 2011
 * @author Fundamentos de Programacion II
 */
public class Calculadora {

    /** Atributo privado donde se almacena los resultados obtenidos. */
    private int memoria ;

    /**
     * Constructor de la clase Calculadora.
     *
     * Parametros: No hay parámetros.
     */
    public Calculadora() {
        this.memoria = 0;
    }

    /**
     * Descripcion: Este metodo realiza la suma de los dos parametros
     *               proporcionados.
     *
     */
    public int suma( int param1 , int param2 ) {
        int resultado = 0; // Almacena el resultado de la suma .

        resultado = param1 + param2 ;
        this.memoria = resultado ; /* Se almacena en memoria.*/
        return resultado ;
    }

    /**
     * Descripcion: Este metodo realiza la suma del valor
     *               proporcionado como par metros con el valor
     *               almacenado en memoria.
     */
    public int suma( int param ) {
        int resultado = 0;

        resultado = param + this.getMemoria();
        this.memoria = resultado ; /* Lo almacenamos en memoria. */
        return resultado ;
    }

    /**
     * Descripcion: Este metodo realiza la division de dos valores
     *               proporcionados como par metros.
     *
     */
}

```

```

public int divide( int param1 , int param2 ) {
    int resultado = 0; // Almacena el resultado de la division.

    resultado = param1 / param2 ;
    this.memoria = resultado ; /* Lo almacenamos en memoria. */
    return resultado ;
}

/**
 * Descripcion: Este metodo calcula el factorial de un numero.
 *
 */
public int factorial( int param ) {
    int resultado = 1;

    for (int i = param; i > 0; i -- ){
        resultado = resultado * i;
    }
    return resultado;
}

/**
 * Descripcion: Este metodo devuelve el valor almacenado en
 * memoria.
 */
public int getMemoria() {
    return this.memoria;
}
}

```

2. Compile el fichero Calculadora.java utilizando las opciones -g y -Xlint. Compruebe qué fichero se ha generado y en qué carpeta. Elimine el archivo que se ha generado en la compilación de Calculadora.java.

Nota: puede usar **-encoding ISO-8859-1** para eliminar los errores en la compilación debido al sistema de codificación del fichero con el código fuente.

3. Compile el fichero Ejercicio01.java desde el directorio padre de fp2, utilizando las opciones -g y -Xlint. Para compilarlo se debe proporcionar toda la ruta hasta el directorio donde se encuentra el fichero .java (./fp2/poo/practical/Ejercicio01.java).

¿Cuántos ficheros .class se han generado?

Mueva el fichero Calculadora.java a otro directorio y elimine todos los archivos .class del directorio donde está ubicado Ejercicio01.java. Vuelve a compilar ¿Por qué sucede este error?

Coloque de nuevo Calculadora.java, en el directorio donde estaba situado.

Incluya la siguiente sentencia import en el fichero Ejercicio01.java y vuelva a compilar.

```
import fp2.poo.practical.Calculadora;
```

Al pertenecer al mismo paquete, no es necesaria esta sentencia import, aunque es aconsejable incluirla.

4. Ejecute la aplicación cuya función principal se encuentra en la clase Ejercicio01, tecleando en la línea de comandos

```
java fp2.poo.practical.Ejercicio01
```

5. Cree el archivo Ejercicio01.jar en el que se encuentren todos los ficheros .class necesarios para la ejecución de la función principal que se encuentra en la clase Ejercicio01. Esto lo podrá realizar mediante los siguientes comandos:

```
jar cvfe Ejercicio01.jar fp2.poo.practica1.Ejercicio01 ./fp2/poo/practica1/Ejercicio01.class ./fp2/poo/practica1/Calculadora.class
```

Para la generación del .jar. Y

```
java -jar Ejercicio01.jar
```

para su ejecución.

6. Elimine todos los ficheros .class del directorio fp2/poo/practical1.

7. Ejecute la aplicación que se encuentra almacenada en el fichero Ejercicio01.jar. Observe el resultado.

8. Observe el siguiente fichero makefile que automatiza las tareas de compilación de los ficheros .class y la construcción del fichero Ejercicio01.jar, proporcionado en esta práctica.

```
#  
#   Makefile de ejemplo para la compilación, creación del jar y ejecución  
  
# CLASEPPAL es el nombre de la clase que contiene el metodo principal (main)  
CLASEPPAL=Ejercicio01  
  
# CLASEAUX es el nombre de la clase que se esta probando.  
CLASEAUX=Calculadora  
  
RUTACLASE=fp2/poo/practical1/  
  
PRINCIPAL=fp2.poo.practica1.Ejercicio01  
  
ejecutaJ: $(CLASEPPAL).jar  
        java -jar $(CLASEPPAL).jar  
  
$(CLASEPPAL).jar: $(RUTACLASE)$(CLASEPPAL).class $(RUTACLASE)$(CLASEAUX).class  
        jar cvfe $(CLASEPPAL).jar $(PRINCIPAL) $(RUTACLASE)$(CLASEPPAL).class  
        $(RUTACLASE)$(CLASEAUX).class  
  
$(RUTACLASE)$(CLASEPPAL).class: $(RUTACLASE)$(CLASEPPAL).java  
        javac -g -encoding ISO-8859-1 -Xlint $(RUTACLASE)$(CLASEPPAL).java  
  
$(RUTACLASE)$(CLASEAUX).class: $(RUTACLASE)$(CLASEAUX).java  
        javac -g -encoding ISO-8859-1 -Xlint $(RUTACLASE)$(CLASEAUX).java
```

9. Utilice la herramienta make para comprobar que el fichero makefile del apartado 8 funciona correctamente. Por ejemplo, pruebe a realizar sucesivas ejecuciones. Pruebe la opción -B con make.

4.2. Organización de los ficheros.

Se propone, a partir del ejercicio anterior, construir un entorno de trabajo que le permita trabajar de manera ordenada con el código.

1. Descargue el fichero **p1MejoradaSegundaParteDeLaPractica.zip** de la plataforma de enseñanza virtual y descomprimalo. Al descomprimirlo aparece el directorio **p1Mejorada**.
2. En este directorio aparecen los siguientes subdirectorios:
 - **src** En este directorio se va a almacenar todo el código fuente que se escriba (ficheros .java) durante las sesiones prácticas.
 - **bin** En este directorio se van a almacenar todos los ficheros bytecode que se genere (ficheros .class) a partir de los ficheros almacenados en los directorios **src**.
 - **jar** En este directorio se van a almacenar todos los ficheros .jar que se generen.
3. En el directorio **src** de **p1Mejorada** aparecen subdirectorios, y en **practical** el código fuente (.java). Es decir, en el directorio **p1Mejorada/src/fp2/poo/practical** aparezcan los ficheros **Calculadora.java** y **Ejercicio01.java**.
4. Elimine todos los ficheros .class y .jar si aparecen generados.
5. Partiendo del fichero makefile proporcionado anteriormente, crear un fichero makefile en el directorio **p1Mejorada**, que automatice las tareas de compilación, creación del jar y ejecución, generando los ficheros .class en el subdirectorio ./bin, y el fichero .jar en el directorio ./jar.
6. Ejecute la aplicación que se encuentra almacenada en el fichero **Ejercicio01.jar**.

Nota: el makefile para la parte de p1Mejorada se proporciona en un segundo fichero comprimido en la plataforma de enseñanza virtual en la carpeta correspondiente a la práctica 1.

Trabajo a entregar

- Cree un directorio de nombre su UVUS.
- Cree en él los directorios **src**, **bin** y **jar**.
- Cree en **src** el subdirectorio asociado al paquete **fp2.poo.practicalXXX**, donde **XXX** es el UVUS del alumno, y que contenga el fichero **HolaMundo.java**. Este fichero deberá mostrar por la salida estándar (pantalla) el texto: “**Hola Mundo**”.
- Haga un **makefile** para realizar la compilación del código, de tal forma que al compilar el archivo fuente coloque los ficheros **.class** a partir del directorio **bin**.
- Genere el **.jar** en el directorio **jar**, y ejecútelo.
- Comprima el directorio y entréguelo en la tarea correspondiente a la práctica 1.



MATRICES

1. OBJETIVO

El objetivo de esta práctica es proporcionar estructuras y clases que están presentes en el funcionamiento básico de la mayoría de los programas escritos en el lenguaje de programación Java. Esta práctica muestra el uso de las matrices en el lenguaje de programación Java. Las matrices dentro de este lenguaje son objetos.

2. DIRECTORIO DE LA PRÁCTICA

El código fuente de esta práctica se deberá colocar en el subdirectorio **fp2.poo.matrices** del directorio **src**. Coloque en él todos los programas de esta práctica. Descargue el código de los ejemplos de esta práctica de la plataforma virtual, y descomprímalo.

3. Introducción. Conceptos Básicos.

Una matriz es un grupo de variables del mismo tipo a las que se hace referencia con el mismo nombre. Se pueden crear matrices de cualquier tipo y pueden tener una o más dimensiones. A un elemento específico de una matriz se accede por su *índice*. Las matrices ofrecen un medio de agrupar información relacionada.

Nota: Las matrices en Java funcionan de manera diferente a como lo hacen en el lenguaje C.

3.1 Matrices unidimensionales.

Una **matriz unidimensional** es, básicamente, una lista de variables del mismo tipo. Para crear una matriz, primero se tiene que crear una variable matriz del tipo deseado. La forma general de una declaración de matriz unidimensional es.

```
tipo nombreMatriz [];
```

Donde **tipo** declara el tipo básico de la matriz, que determina el tipo de cada elemento de la misma. El siguiente ejemplo muestra la declaración de una matriz llamada **diasDelMes** con el tipo "matriz de int":

```
int diasDelMes [];
```

Aunque esta declaración establece que **diasDelMes** es una variable matriz, realmente no existe ninguna matriz. De hecho, el valor de **diasDelMes** es asignado a **null**, que representa una matriz sin valor.

Para unir **diasDelMes** con una matriz de enteros "real", se debe reservar espacio utilizando **new** (operador de reserva de memoria) y asignándolo a **diasDelMes**.

El operador **new** tiene la siguiente forma general cuando aparece con matrices unidimensionales:

```
nombreMatriz = new tipo[tamaño];
```

Donde **tipo** especifica el tipo de los datos que van a ser asignados, **tamaño** especifica el número de elementos de la matriz y **NombreMatriz** es la variable matriz a la que se asigna la nueva matriz. Cuando se utiliza **new** para reservar espacio para una matriz, es necesario especificar el tipo y el número de elementos que se quieren reservar. Los elementos de una matriz reservada con **new** son inicializados automáticamente a cero.

En este ejemplo se reserva una matriz de enteros con 12 elementos y se asigna a **diasDelMes**:

```
diasDelMes = new int [12];
```

Cuando se ejecuta esta sentencia, **diasDelMes** se refiere a una matriz de 12 enteros. Además, todos los elementos de la matriz estarán inicializados a cero.

En resumen, la obtención de una matriz es un proceso en **dos pasos**.

1. En primer lugar es necesario declarar una variable del tipo de matriz deseado.
2. En segundo lugar, utilizando el operador **new**, se tiene que reservar la memoria en la que estará la matriz y se asigna a la variable matriz. En Java, la memoria necesaria para cada matriz se reserva dinámicamente.

Una vez que se ha reservado memoria para una matriz, se puede acceder a un elemento específico de la misma especificando su índice entre corchetes (`[]`). Al primer elemento de una matriz le corresponde el índice cero. Por ejemplo, esta sentencia asigna el valor 28 al segundo elemento **diasDelMes**.

```
diasDelMes[1] = 28;
```

La siguiente línea escribe el valor almacenado en el elemento con índice 3.

```
System.out.println(diasDelMes[3]);
```

Para resumir de manera práctica todo lo que hemos visto hasta ahora, el siguiente programa crea una matriz con los números de los días de cada mes.

```

/*
 * Fichero: EjemploMatriz.java
 *
 * Fundamentos de Programación II. GIT.
 * Departamento de Ingeniería Telemática
 * Universidad de Sevilla
 */

package fp2.poo.matrices;

/**
 * Descripción: Esta es una clase de ejemplo para utilizar
 * una matriz en Java.
 *
 * @version versión 1.0 Abril 2011
 * @author Fundamentos de Programación II
 */
public class EjemploMatriz {

    /*
     * Descripción: Método main que declara una variable matriz,
     * realiza la instanciación y accede a sus elementos.
     */
    public static void main(String args[ ]) {
        int diasDelMes [] = null;

        diasDelMes = new int [12];
        diasDelMes[0] = 31;
        diasDelMes[1] = 28;
        diasDelMes[2] = 31;
        diasDelMes[3] = 30;
        diasDelMes[4] = 31;
        diasDelMes[5] = 30;
        diasDelMes[6] = 31;
        diasDelMes[7] = 31;
        diasDelMes[8] = 30;
        diasDelMes[9] = 31;
        diasDelMes[10] = 30;
        diasDelMes[11] = 31;
        System.out.println("Abril tiene " + diasDelMes[3] + " días.");
    }
}

```

Cuando se ejecuta este programa se imprime el número de días de Abril. Al primer elemento de una matriz le corresponde el índice cero, por lo que el número de días de Abril es `diasDelMes[3]` o, lo que es lo mismo 30.

Ejercicios.

1. Utilice el siguiente makefile para probar el código de `EjemploMatriz.java` (este makefile se proporciona con los ficheros de la práctica).

```

#
# Makefile de ejemplo para la compilación, creación del jar y ejecución
#
CLASE=EjemploMatriz
RUTACLASE=fp2/poo/matrices/

DIRSRC=../src/
DIRBIN=../bin/
DIRJAR=../jar/
PRINCIPAL=fp2.poo.matrices.EjemploMatriz

ejecutaJ: $(DIRJAR)$(CLASE).jar
    java -jar $(DIRJAR)$(CLASE).jar

$(DIRJAR)$(CLASE).jar: $(DIRBIN)$(RUTACLASE)$(CLASE).class
    jar cvfe $(DIRJAR)$(CLASE).jar $(PRINCIPAL) -C $(DIRBIN) $(RUTACLASE)$(CLASE).class

$(DIRBIN)$(RUTACLASE)$(CLASE).class: $(DIRSRC)$(RUTACLASE)$(CLASE).java
    javac -g -Xlint -d $(DIRBIN) $(DIRSRC)$(RUTACLASE)$(CLASE).java

```

2. Modifique el fichero **EjemploMatriz.java** para mostrar el número de días que tiene el mes de Enero y Diciembre.

3.2 Más sobre matrices.

Como se muestra a continuación, es posible combinar la declaración de una variable matriz con la operación de reservar memoria.

```
int diasDelMes [] = new int [12];
```

Esta es la forma que normalmente se utiliza en los programas Java. Las matrices se pueden inicializar cuando son declaradas. El mecanismo es similar al que se utiliza para inicializar los tipos simples. Un **inicializador de matriz** es una lista de expresiones separadas por comas y entre llaves ({}). Las comas separan los valores de los elementos de la matriz. Se creará una matriz lo suficientemente grande para contener el número de elementos que se especifiquen en la inicialización de la matriz. No es necesario utilizar el operador **new**.

Por ejemplo, para almacenar el número de días de cada mes, el siguiente código crea una matriz de enteros.

```
//versión mejorada
class EjemploMatriz {
    public static void main(String args[ ]) {
        int diasDelMes [] =
            {31,28,31,30,31,30,31,31,30,31,30,31};
        System.out.println("Abril tiene"+ diasDelMes[3]+"días");
    }
}
```

Java comprueba estrictamente que no se intenta almacenar o hacer referencia accidentalmente a valores que estén fuera del rango de la matriz. El intérprete de Java se asegurará de que todos los índices de la matriz están dentro del rango correcto. En este aspecto, Java es diferente a C/C++, ya que estos lenguajes no realizan comprobaciones en tiempo de ejecución. Por ejemplo, el

intérprete de Java comprobará el valor de cada índice de **díasDelMes** para asegurarse de que está comprendido entre 0 y 11, ambos inclusive. Si se intenta acceder a elementos que están fuera del rango de la matriz, a números negativos o a números positivos mayores que la longitud de la matriz, se producirá un error en tiempo de ejecución.

Ejercicios.

3. Modifique el código fuente para que realice la inicialización de la matriz en una misma línea de código tal y como aparece en la versión mejora de arriba.
4. Modifique el fichero **EjemploMatriz.java** para que acceda a una posición que esté fuera del rango de la matriz.

A continuación se muestra otro ejemplo que utiliza una matriz unidimensional. Este programa calcula la media de un conjunto de valores. Este código se proporciona en el código descargado de la Enseñanza Virtual.

```
//Calcula la media de los valores de una matriz
class Media {
    public static void main(String args[ ]) {
        double nums[] = {10.1,11.2, 12.3,13.4,14.5};
        double result = 0;
        for (int i = 0; i<5; i++)
            result += nums[i];
        System.out.println("La media es"+ result/5);
    }
}
```

3.2 Matrices Multidimensionales.

En Java, las matrices multidimensionales realmente son matrices de matrices. Éstas, como se podría esperar, se parecen y funcionan como las matrices multidimensionales habituales. Sin embargo, como veremos, hay diferencias sutiles.

Para declarar una variable del tipo matriz multidimensional, es necesario especificar cada índice adicional utilizando otra pareja de corchetes ([]).

Por ejemplo, la siguiente línea declara la variable **dosDim** como matriz bidimensional.

```
int dosDim[][] = new int[4][5];
```

Este código reserva una matriz de 4 por 5 y la asigna a la variable **dosDim**. Internamente, esta matriz se implementa como una **matriz de matrices de int**. Conceptualmente, esta matriz se parece a la mostrada en la siguiente figura.

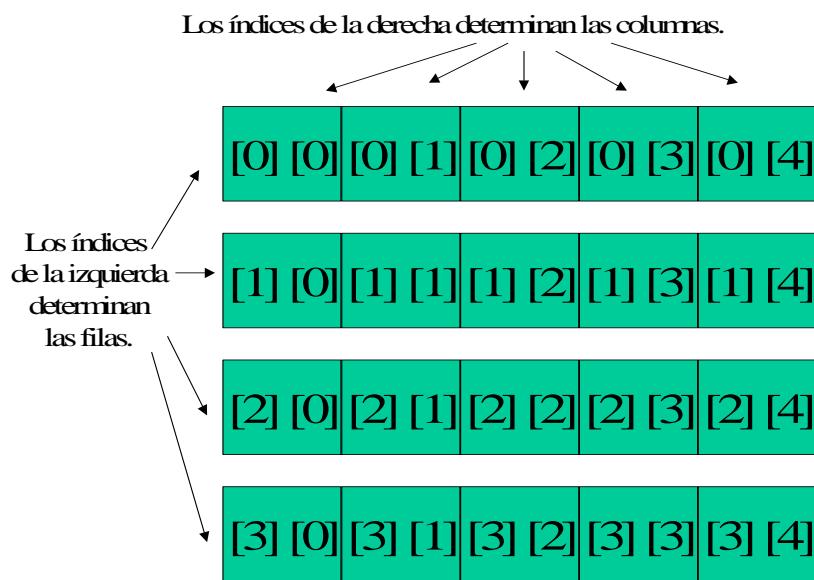


Figura 1. Matriz bidimensional de 4x5.

El siguiente programa enumera los elementos de la matriz de izquierda a derecha y de arriba abajo y después escribe estos valores.

```

/*
 *  Fichero: MatrizBidimensional.java
 *
 *  Fundamentos de Programacion II. GIT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *
 */

package fp2.poo.matrices;

/**
 * Descripcion: Esta es una clase mantiene una matriz bidimensional
 *
 * @version versi n 1.0 Abril 2011
 * @author Fundamentos de Programacion II
 */
public class MatrizBidimensional {

    /** n mero de filas de la matriz bidimensional */
    private int filas      = 0;

    /** n mero de columnas de la matriz bidimensional */
    private int columnas   = 0;

    /** la matriz bidimensional */
    private int biDim[][] = null;

    /**
     * Descripci n: Constructor de la clase.
     *           Inicializa los valores
     * @param filas numero de filas
     * @param columnas numero de columnas
     */
    public MatrizBidimensional( int filas, int columnas) {
        this.biDim      = new int[filas][columnas];
        this.filas      = this.biDim.length;
        this.columnas   = columnas;
        this.inicializacion();
    }

    /**
     * Descripcion: Inicializa los valores de la matriz.
     *
     * @return Nada.
     */
    private void inicializacion(){
        for (int i = 0, k = 0; i < this.filas ; i++){
            for (int j = 0; j < this.columnas; j++) {
                this.biDim [i][j] = k;
                k++;
            }
        }
    }

    /**
     * Descripcion: Muestra el contenido de la matriz.
     *
     * @return Nada.
     */
    public void imprimeMatriz() {
        for (int i = 0; i < biDim.length; i++) {
            for (int j = 0; j < this.biDim[i].length; j++) {
                System.out.print("\t" + this.biDim[i][j]);
            }
            System.out.println();
        }
    }

    /**
     * Descripcion: Metodo getter.
     *
     * @return Devuelve el n mero de filas de la matriz.
     */
    public int getNumFilas(){
        return this.filas;
    }

    /**
     * Descripci n: M todo getter.
     *
     * @return Devuelve el n mero de columnas de la matriz.
     */
}

```

El programa genera la siguiente salida:

```
# java -classpath ./bin fp2.poo.matrices.Bidimensional
      0      1      2      3      4
      5      6      7      8      9
     10     11     12     13     14
     15     16     17     18     19
```

• Cuando se reserva memoria para una matriz multidimensional, sólo es necesario especificar la memoria que necesita la primera dimensión, es decir, la que está más a la izquierda.

• Después se puede reservar la memoria para las otras dimensiones de manera independiente. Por ejemplo, el siguiente código reserva memoria para la primera dimensión de **biDim** cuando se declara y después, reserva memoria para la segunda dimensión:

```
int biDim[][] = new int[4][];
biDim[0] = new int [5];
biDim[1] = new int [5];
biDim[2] = new int [5];
biDim[3] = new int [5];
```

Nota: Al hacerlo de esta forma no es necesario utilizar el mismo número de elementos para cada dimensión ya que la longitud de cada matriz se puede establecer de forma independiente. Aunque no se recomienda utilizar matrices multidimensionales donde la longitud de cada matriz sea diferente de manera generalizada.

También es posible inicializar matrices multidimensionales. Para hacer esto, simplemente es necesario encerrar entre llaves el inicializador de cada dimensión.

El siguiente programa crea una matriz en la que cada elemento contiene el producto del índice de su columna por el índice de su fila. Observe que, además de valores literales, puede utilizar expresiones dentro de la inicialización de una matriz.

```
class DosDim {
    public static void main(String args[ ]) {
        int m[][] = {
            { 0*0, 1*0, 2*0, 3*0},
            { 0*1, 1*1, 2*1, 3*1},
            { 0*2, 1*2, 2*2, 3*2},
            { 0*3, 1*3, 2*3, 3*3},
        };
        int i, j;

        for (i = 0; i<4; i++) {
            for (j = 0; j<4; j++) {
                System.out.print(m[i][j] + " ");
            }
        }
    }
}
```

```

        System.out.println();
    }
}
}

```

Cuando se ejecuta este programa se obtiene la siguiente salida.

```

0 0 0 0
0 1 2 3
0 2 4 6
0 3 6 9

```

Como se puede observar, cada fila de la matriz está inicializada con los valores especificados en las listas de inicialización.

Sintaxis alternativa para la declaración de una matriz

Aunque la sintaxis para declarar una matriz que se recomienda es la vista anteriormente se tiene esta otra forma: **tipo[] nombreMatriz;**. En este caso, los corchetes siguen al tipo y no al nombre de la matriz. Las siguientes dos declaraciones son equivalentes:

```

int a1[] = new int[3];
int[] a1 = new int[3];

```

Las siguientes declaraciones también son equivalentes:

```

char dosDim[][] = new char[3][4];
char[][] dosDim = new char[3][4];

```

Ejercicios.

1. El ejemplo de la clase bidimensional está en el mismo paquete que la matriz unidimensional. Implemente un makefile para realizar su compilación.
2. Genere el fichero .jar teniendo en cuenta que se generan dos ficheros .class.
3. Ejecute el código.

Trabajo a entregar.

1. Cree un directorio de nombre su UVUS.
2. Cree en él los directorios src, bin y jar.
3. Cree en src el subdirectorio asociado al paquete **fp2.poo.practica2XXX** donde **XXX** es el UVUS del alumno.
4. Use el código de la clase **Cuenta** que se muestra a continuación para crear desde el método **main** de la clase **CuentaPrincipal** (en el paquete **fp2.poo.practica2XXX**) una tabla de 20 objetos del tipo **Cuenta**. Recuerde que una vez reservada la matriz de 20 elementos hay que instanciar cada uno de los objetos de tipo **Cuenta**. Inserte estos objetos de tipo **Cuenta** en la tabla creada.
5. Realice los comentarios, tabulación y estructura a la nueva clase creada según lo especificado en la práctica 0.
6. Compile ambas clases y genere el fichero jar en el makefile, usando la estructura de directorios utilizada anteriormente (**src**, **bin** y **jar**).

```

/*
 *  Fichero: Cuenta.java
 *
 *  Fundamentos de Programacion II. GIT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *
 */

package fp2.poo.utilidades;

/**
 * Descripcion: Esta es una clase que representa una cuenta bancaria.
 *              Mantiene una asociacion entre un usuario (nombre) y.
 *              su saldo (saldo).
 *
 * @version version 1.0 Abril 2011
 * @author Fundamentos de Programacion II
 */
public class Cuenta {

    /** nombre es el atributo asociado al nombre de un usuario */
    private String nombre;

    /** saldo es el atributo asociado al dinero de una cuenta de usuario */
    private double saldo;

    /*
     * Descripcion: Constructor de la clase.
     */
    public Cuenta ( String nombre, double saldo ) {
        this.nombre = nombre;
        this.saldo = saldo ;
    }

    /*
     * Descripcion: Este metodo configura el atributo nombre a un valor.
     */
    public void setNombre( String nombre ) {
        this.nombre = nombre;
    }

    /*
     * Descripcion: Este metodo configura el atributo saldo a un valor.
     */
    public void setSaldo( double saldo ) {
        this.saldo = saldo ;
    }

    /*
     * Descripcion: Este metodo obtiene el atributo nombre.
     */
    public String getNombre( ) {
        return this.nombre ;
    }

    /*
     * Descripcion: Este metodo obtiene el valor del atributo saldo.
     */
    public double getSaldo( ) {
        return this.saldo;
    }
}

```



ENTORNO DE PROGRAMACIÓN (II)

OBJETIVO

En esta práctica se presenta la herramienta `jdb` para la depuración. En esta práctica se realiza la implementación de una interfaz para la depuración del código.

1. Depurador (`jdb`).

El programa `jdb` es una herramienta muy útil para la depuración de aplicaciones escritas en Java. La sintaxis de este programa es la siguiente:

`jdb [opciones] clase [argumentos]`

donde:

- **[opciones]** modifican el comportamiento del depurador. Algunas de las opciones más frecuentes son las siguientes:

<code>-classpath ruta</code>	Opción encargada de indicar al depurador la ruta a los archivos <code>.class</code> que se desean depurar. El contenido de esta opción sobrescribe el contenido de la variable de entorno <code>CLASSPATH</code> . En caso de que no se haya especificado la mencionada variable y tampoco se utilice esta opción, la ruta utilizada será el directorio de trabajo <code>(.)</code> . Cada uno de los caminos se indicarán separados por <code>:</code> .
------------------------------	---

<code>-sourcepath ruta</code>	Opción encargada de indicar al depurador la ruta donde se encuentran los ficheros de código fuente. Cada una de las rutas se indicarán separados por <code>:</code> .
-------------------------------	---

- **clase** indica el nombre de la clase en la que se desea empezar la depuración.
- **[argumentos]** indica los argumentos que se desean pasar al método `main` de la clase indicada anteriormente.

A continuación se van a mostrar los comandos principales que se pueden utilizar dentro del depurador jdb:

- Orden **help**:

La orden **help** se encarga de mostrar la lista de comandos válidos dentro del depurador.

- Orden **run (run [class [args]])**:

La orden **run** se encarga de iniciar la ejecución de la aplicación Java que se está depurando, comenzando la ejecución en la clase **main** de la aplicación.

- Orden **cont**:

La orden **cont** indica al depurador que se desea continuar la ejecución de la aplicación desde el breakpoint.

- Órdenes **stop** y **clear**:

La orden **stop** nos permite fijar puntos de parada en nuestro programa. La sintaxis de esta orden es:

stop at clase:linea Crea un punto de parada en la línea **linea** del fichero que contiene el código fuente la clase **clase** (incluyendo el paquete al que pertenece).

stop in metodo Crea un punto de parada en el método **metodo**. El método generalmente se indicará de la forma **clase.metodo** (incluyendo también el paquete al que pertenece).

stop in clase.<init> Crea un punto de parada en el constructor de la clase **clase** (incluyendo el paquete al que pertenece).

stop Muestra los puntos de parada establecidos hasta el momento.

La orden **clear** nos permite eliminar puntos de parada fijados anteriormente.

- Órdenes **catch e ignore**:

La orden **catch** nos permite indicar al depurador que este debe detener la ejecución del programa en el caso de que se lance la excepción que le indiquemos. La sintaxis de esta orden es **catch excepcion** (incluyendo el paquete al que pertenece la excepción **excepcion**). De forma opuesta, la orden **ignore** nos permite anular el efecto de una orden **catch** previa. Las excepciones en Java se verán en una práctica posterior.

- Orden **step**:

La orden **step** nos permite ejecutar la línea de código en la que estamos detenidos, tras lo cual el control vuelve el depurador. En caso de que la línea sea una llamada a un método se ejecutarán todas las sentencias del método una a una mediante sucesivas llamadas a **step**.

- Orden **stepi**:

Ejecuta solo una instrucción.

- Orden **next**:

La orden **next** nos permite ejecutar la siguiente línea de código en la función en la que estamos detenidos, tras lo cual el control vuelve al depurador. En caso de que la línea sea una llamada a un método se ejecutará con una sola ejecución del comando **next**.

- Orden **print**:

A través de la orden **print** podemos obtener información sobre las variables de instancia de los objetos de nuestra aplicación. En el caso de utilizar esta orden para pedir información sobre un objeto, esta nos devolvería una breve descripción del objeto. Por último, esta orden soporta la mayoría de las expresiones que se pueden utilizar en Java.

Por ejemplo:

```
print MiClase.miAtributoEstatico
print miObjeto.miAtributo
print i + j + k (i, j, k son tipos primitivos y ambos son atributos o variables locales)
print miObjeto.miMetodo() (if myMethod returns a non-null)
print new java.lang.String("Hello").length()
```

- Orden **dump**:

A través de la orden **dump** podemos obtener información de cada una de las variables de un objeto.

En el caso de utilizar la orden **dump** con variables de instancia, el funcionamiento es igual que el de la orden **print**. Además, esta orden, al igual que la orden anterior, soporta la mayoría de las expresiones que se pueden utilizar en Java.

Para objetos, imprime el valor actual de cada campo definido en el objeto. Incluidos los campos estáticos y las variables de instancia.

El comando **dump** soporta las mismas expresiones que el comando **print**.

- Orden **eval <expr>**:

Evalúa una expresión (igual que **print**).

- Orden **set** (Sintaxis **set <lvalue> = <expr>**):

Asigna un nuevo valor al elemento campo/variable/array.

- Orden **where**:

La orden **where** nos permite conocer el contenido de la pila del programa que estamos depurando.

- Orden **up** y **down** (**up [n frames]** y **down [n frames]**):
Con el comando up y down, selecciona que trama de la pila es la actual.
- Orden **list**:
La orden lista muestra por pantalla el código fuente que estamos depurando.
- Orden **quit**:
La orden quit finaliza la ejecución del depurador.
- Orden **locals**:
Imprime todas las variables en la trama actual de la pila.
- Orden **classes**:
Muestra las clases actualmente conocidas
- Orden **class <class id>**:
Muestra detalles de la clase nombrada.
- Orden **methods <class id>**:
Lista los métodos de una clase.
- Orden **fields <class id>**:
Lista los campos de una clase.
- Orden **!!**:
Repite el ultimo comando.
- Orden **<n> <comando>**:
Repite <n> veces el comando <comando>.
- Orden **read <fichero_de_comandos>**:
Lee un fichero que contiene los comandos a ejecutar.

Ejercicios propuestos.

1. Cree un directorio de nombre su uvus, con los subdirectorios **src**, **bin** y **jar**, en este directorio deberá crear el Makefile para realizar la compilación, creación del jar y ejecución del código.
2. Descargue el código de la plataforma y coloquelo en el directorio creado anteriormente (de nombre su uvus). Escriba el código Java de la clase **Usuario.java** en el paquete **fp2.poo.practica3XXX**, siendo **XXX** el uvus del alumno, (cree los subdirectorios asociados a este paquete en el directorio **src**) para que implemente los métodos de la interfaz **Persona.java**. Implemente al menos un constructor. Cuando compile esta interfaz utilice la opción apropiada para dejar el bytecode en el directorio **./bin**.

```
/*
 *   Fichero: Persona.java
 *
 *   Fundamentos de Programacion II. GITT.
 *   Departamento de Ingenieria Telematica
 *   Universidad de Sevilla
 *
 */

package fp2.poo.practica3;

/**
 * Descripcion: Esta una interfaz con los metodos de Persona.
 *
 * @version version 1.0 Mayo 2011
 * @author Fundamentos de Programacion II
 */
public interface Persona {

    public String getNombre( );
    public String getPrimerApellido( );
    public String getSegundoApellido( );
    public String getDNI( );
    public String getDomicilio( );

    public void setNombre( String nombre );
    public void setPrimerApellido( String primerApellido );
    public void setSegundoApellido( String segundoApellido );
    public void setDNI( String dni );
    public void setDomicilio( String domicilio );
}
```

3. Escriba el código Java de la clase **Main.java** (en el paquete **fp2.poo.practica3**) que contenga el método **main**, e instancie un objeto de la clase **Usuario** mediante el constructor implementado. Obtenga el valor de los atributos y sáquelos por la salida estándar mediante el método **System.out.println**. Use una variable referencia a la interfaz (**Persona**) para referirse a un objeto de la clase que implementa (**Usuario**).
4. Escriba el fichero **makefile** para realizar la compilación, y la creación del fichero **Usuario.jar**, de las clases implementadas y de la interfaz. Utilice la estructura de directorios de las prácticas anteriores (código en el directorio **src**, un directorio **jar** con el código empaquetado y el código compilado en **bin**).

Nota: Para evitar **warning** en la compilación debido a los acentos de los comentarios compile con la opción **-encoding ISO-8859-1**.

5. Ejecute el depurador **jdb**, de la siguiente forma:

```
jdb -classpath ./jar/Usuario.jar -sourcepath ./src fp2.poo.practica3.Main
```

6. Ponga un punto de parada al comienzo del método **main** de la clase **Main** del paquete **fp2.poo.practica3**.
7. Añada otro punto de parada en el constructor de la clase **Usuario**, en el método **main**.
8. Compruebe de que realmente tiene el punto de parada.
9. Ejecute el programa.
10. Vuelva a realizar la misma operación escribiendo en un fichero de texto con los comandos para realizar los puntos de parada, y ejecútelo con el comando **read**.
11. Hay un listado del código del programa en el punto en el que se ha parado.
12. Mire el contenido de la traza de la pila y las variables locales.
13. Continúe la ejecución hasta el constructor de la clase **Usuario**.
14. Ejecute línea a línea.
15. Mire el contenido de la traza de la pila y las variables locales.
16. Muévase en la trama de pila para obtener el contexto del método **main**. Muestre las variables locales del método.
17. Muévase en la trama de pila para obtener el contexto del constructor de **Usuario**. Muestre las variables locales del método.
18. Continúe la ejecución del programa.

Trabajo a entregar.

1. Realice los comentarios, tabulación y estructura a la nueva clase creada (**Usuario.java** y **Main.java** del ejercicio realizado en esta práctica) según lo especificado en la práctica 0.
2. Comprima el directorio de nombre su UVUS que contiene todo el código realizado y el makefile realizado en esta práctica en el archivo **UVUS.zip**, y realice la entrega.



EXCEPCIONES

1. OBJETIVO

El objetivo de esta práctica es mostrar la gestión de condiciones excepcionales en los lenguajes de programación orientados a objeto mediante el uso de excepciones, y de forma específica el lenguaje de programación Java. Para ello se muestra qué es una excepción, cómo se captura y cómo se lanza desde un programa. Esta práctica presenta cómo crear excepciones propias de la aplicación, y se muestra la forma de tratarlas en el depurador jdb.

2. DESCRIPCIÓN

Las excepciones son situaciones anormales que suceden cuando se provoca un fallo durante la ejecución de un programa. En los lenguajes de programación que no tienen gestión de excepciones, los errores se gestionan normalmente utilizando códigos de error. La gestión de excepciones de Java es una solución más directa para la gestión de errores en tiempo de ejecución y con un enfoque orientado a objetos.

Por ejemplo, se producen excepciones en un programa cuando se hace una división de un entero por cero, o cuando se intenta acceder al elemento de la posición 4 de un vector cuando el vector sólo tiene 2 elementos.

Las excepciones en Java son *objetos* que describen una condición excepcional, es decir, un error que se ha producido en un fragmento de código. Cuando surge una condición excepcional, se crea un objeto que representa la excepción y se *lanza* la excepción avisando de que se ha producido una situación especial.

El método que recibe la excepción puede elegir *capturarla* y gestionarla él mismo o no. En algún punto del programa, la excepción es capturada y procesada. Las excepciones pueden ser generadas por el **intérprete** de Java o pueden ser generadas **por el propio código**.

Excepciones no capturadas

Antes de aprender cómo se gestionan las excepciones en los programas en Java, es útil ver qué ocurre cuando no se gestionan. Los errores de ejecución pueden presentarse en cualquier lenguaje de programación. Empecemos por un primer ejemplo en el lenguaje de programación C. Descargue el código de la plataforma virtual para realizar la práctica 4 y descomprímalo. Se proporciona el siguiente código que intencionadamente incluye un error de división por cero.

```
#include <stdio.h>

int main() {
    int d = 0;
    int a = 0;

    a = 42 /d;
    printf("Este codigo no se ejecuta nunca\n");

    return 0;
}
```

Practica4Ejercicio00.c

Utilice la herramienta make para compilar y ejecutar el código de la siguiente forma y observe el resultado:

```
make -f makeP4 compila_c00
make -f makeP4 ejecuta_c00
```

Este mismo ejemplo se puede realizar sobre el lenguaje de programación Java.

```
package fp2.poo.practica4;

public class Practica4Ejercicio01 {
    public static void main( String args[] ) {
        int d = 0;
        int a = 42 /d;
        System.out.println( "Este codigo nunca se ejecuta" );
    }
}
```

Practica4Ejercicio01.java

Utilice la herramienta make para compilar los ejemplos que se van a ver a continuación, de la siguiente forma:

```
make -f makeP4 compilaJava
```

y ejecute el código de este primer ejemplo en Java de la siguiente forma:

```
make -f makeP4 ejecuta01
```

Nota: a partir de ahora cada ejecución de los ejemplos se realizará mediante **make -f makeP4 ejecutaxx** con **xx** que indica el número del ejercicio que se está probando.

Cuando el intérprete de Java detecta el intento de dividir por cero, construye un nuevo objeto del tipo `Exception` y *lanza* la excepción. Esto hace que se detenga la ejecución del programa, ya que una vez que se ha lanzado una excepción, tiene que ser *capturada* por el gestor de excepciones y tratada inmediatamente. En este ejemplo, no hemos proporcionado un *gestor de excepciones propio*, por lo que la excepción será capturada por el gestor proporcionado por el

intérprete de Java. Cualquier excepción que no es capturada por el programa será tratada por el gestor por defecto, quien muestra un mensaje describiendo la excepción, imprime el trazado de la pila del lugar donde se produjo la excepción y termina el programa.

Esta es la salida generada cuando este ejemplo se ejecuta en el intérprete de Java.

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
at fp2.poo.practica4.Practica4Ejercicio01.main(Practica4Ejercicio01.java:22)
```

Observe que en el trazado de la pila se incluye:

- el nombre de la clase, **fp2.poo.practica4.Practica4Ejercicio01**;
- el nombre del método, **main()**;
- el nombre del archivo **Practica4Ejercicio01.java**,
- y el número de la línea, **22**.

Observe también que el tipo de la excepción que se ha lanzado es una Exception llamada **ArithmetricException**, que describe más específicamente el tipo de error que se ha producido. Java proporciona distintos tipos de excepciones que coinciden con las clases de errores que se pueden producir durante la ejecución.

La traza de la pila siempre muestra la secuencia de llamadas a métodos en el momento del error. Esta es otra versión del programa anterior que introduce el mismo error pero en un método distinto del **main()** .

```
package fp2.poo.practica4;

public class Practica4Ejercicio02 {

    static void metodo() {
        int d = 0;
        int a = 10/d;

        System.out.println("Este codigo nunca se ejecuta");
    }

    public static void main ( String args[] ) {
        Practica4Ejercicio02.metodo();

        System.out.println( "Este codigo nunca se ejecuta" );
    }
}
```

Practica4Ejercicio02.java

Ejecute el programa y observe la traza de la pila que en este caso incluye el método estático. La pila de llamadas es bastante útil para depurar ya que muestra los pasos que conducen al error.

Dado el siguiente ejemplo

```
package fp2.poo.practica4;

public class Practica4Ejercicio03 {
    public static void main(String[] args) {
        String str = null;

        str.length();
        System.out.println("Este código nunca se ejecuta");
    }
}
```

Practica4Ejercicio03.java

Ejecute el programa, e indique qué excepción se produce.

Captura de excepciones

Aunque el gestor de excepciones proporcionado por el intérprete de Java es útil para depurar, normalmente se deseará gestionar las excepciones que se produzcan en los programas, obteniendo así dos ventajas, por un lado, permite determinar el error y, por otro, evita que el programa termine automáticamente. La mayoría de los usuarios estarían cuando menos confusos, si el programa terminara su ejecución e imprimiese una traza de la pila cuando se produjese un error. Afortunadamente, es bastante fácil evitar esto.

Para protegerse de los errores de ejecución y poder gestionarlos, es necesario escribir el código que se quiere controlar dentro de un bloque **try**. Inmediatamente después del bloque **try** es necesario incluir la cláusula **catch** que especifica el tipo de excepción que se desea capturar.

Esta es la forma general de un bloque de gestión de excepciones:

```
try {
    //bloque de código
} catch (TipoExcepción1 exOb) {
    //gestor de excepciones para TipoExcepción1
} catch (TipoExcepción2 exOb) {
    //gestor de excepciones para TipoExcepción2
} finally {
    //bloque de código que se ejecutará antes de que termine
    //el bloque try
}
```

Tipos de excepciones

Todos los tipos de excepción son subclases de la clase **Throwable** (lanzable). Esta clase está en la parte superior de la jerarquía de clases de excepción. Inmediatamente debajo de **Throwable** hay dos subclases que dividen las excepciones en dos ramas distintas.

1. Una rama está encabezada por la clase **Exception**. Esta clase se utiliza para condiciones excepcionales que los programas de usuario deberían capturar. Esta es la clase de partida de las subclases que utilizaremos para crear nuestros propios tipos de excepción. Una subclase importante de **Exception** es **RuntimeException** (excepción durante la ejecución). Las excepciones de este tipo incluyen cosas del estilo de la división por cero o un índice inválido de una matriz.
2. La otra rama del árbol es la clase **Error**, que define las excepciones que no se suelen capturar en los programas en condiciones normales. Un ejemplo de este tipo de errores puede ser el desbordamiento de una pila. Esta práctica no trata las excepciones de tipo **Error**, porque se crean habitualmente como respuesta a fallos catastróficos que no suelen ser gestionados por los programas.

El siguiente programa incluye un bloque **try** y una cláusula **catch** que procesa la excepción **ArithmeticeException** generada por un error de división por cero.

```
package fp2.poo.practica4;

import java.lang.ArithmeticeException;

public class Practica4Ejercicio04 {
    public static void main (String args[]) {
        int d, a;

        try {
            d = 0;
            a = 42/d;
            System.out.println("Este código nunca se ejecuta");
        } catch ( ArithmeticeException e ){
            System.out.println( "División por cero." + e );
        }
        System.out.println( "Después de catch." );
    }
}
```

Practica4Ejercicio04.java

Ejecute el programa y observe la salida.

Observe que la llamada a **println()** dentro del bloque **try** no se ejecuta. Una vez que se ha lanzado la excepción, el control del programa se transfiere desde el bloque **try** al bloque **catch**. Una vez se ha ejecutado la sentencia **catch**, el control del programa continúa en la siguiente línea del programa que hay después del bloque **try/catch**.

Las sentencias **try** y **catch** forman una unidad. El ámbito de la cláusula **catch** está restringido a aquellas sentencias especificadas en la sentencia **try** que la precede. Una sentencia **catch** no puede capturar una excepción lanzada por otras sentencia **try**, excepto en el caso de sentencias **try** anidadas. Las sentencias que están protegidas por **try** tienen que estar entre llaves, es decir, deben estar dentro de un bloque.

Una sentencia **catch** bien diseñada debería resolver la condición de excepción y continuar como si el error nunca se hubiese producido. Por ejemplo, en el siguiente programa cada iteración del bucle **for** obtiene dos enteros de forma aleatoria; divide uno entre otro y el resultado se utiliza para dividir el valor 12.345. El resultado final se guarda en **a**. Si cualquiera de las operaciones de división provoca un error de división por cero, este error será capturado y se asignará a **a** el valor cero, continuando la ejecución del programa.

```
package fp2.poo.practica4;

import java.util.Random;
import java.lang.ArithmetricException;

public class Practica4Ejercicio05 {
    public static void main (String args[]) {
        int a =0, b =0, c =0;
        Random r = new Random();

        for ( int i = 0; i < 2000; i++ ){
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345/(b/c);
            } catch ( ArithmetricException e ){
                System.out.println("División por cero." + e);
                a = 0; //asigna cero y continua
            }
            System.out.println( "a: " + a );
        }
    }
}
```

Practica4Ejercicio05.java

Ejecute el programa para ver el resultado. Nótese que si el valor de **c** es mayor que el de **b** el resultado de la división entera será cero, y en este caso se lanzará una excepción del tipo **ArithmetricException**.

Descripción de una excepción

La clase **Throwable** sobrescribe el método **toString()** definido por la clase **Object**, devolviendo una cadena que contiene la descripción de una excepción. Se puede mostrar esta descripción simplemente pasando la excepción como argumento de una sentencia **println()**. Por ejemplo, el bloque **catch** del ejemplo anterior se puede escribir de la siguiente forma:

```
    } catch (ArithmetricException e) {
        System.out.println("Excepcion: " + e);
        a = 0; //asigna cero y continua
    }
```

Realice la modificación propuesta al ejemplo y pruébela.

Al realizar esta modificación, cada error de división por cero presentará el siguiente mensaje:

Excepción: java.lang.ArithmetricException: / by zero

Aunque en este contexto esto no tiene un gran valor, la capacidad de mostrar la descripción de una excepción puede ser muy valiosa en otras circunstancias, especialmente cuando estamos experimentando con excepciones o cuando estamos depurando.

Manejando varios tipos de excepciones

En algunos casos, la misma secuencia de código puede activar más de un tipo de excepción. Para gestionar este tipo de situaciones, se pueden especificar dos o más cláusulas **catch**, cada una para capturar un tipo distinto de excepción. Cuando se lanza una excepción, se inspeccionan por orden las sentencias **catch** y se ejecuta la primera que coincide con el tipo de excepción que se ha producido. Despues de ejecutar la sentencia **catch**, no se ejecuta ninguna de las restantes, continuando la ejecución al final del bloque **try/catch**. El siguiente programa gestiona dos tipos de excepción diferente (división por cero - **ArithmetricException** – e índice fuera de límites - **ArrayIndexOutOfBoundsException**):

```
package fp2.poo.practica4;

import java.util.Random;

public class Practica4Ejercicio06 {
    public static void main (String args[]){
        try {
            int a = 0;

            a = args.length;
            System.out.println( "a: " + a );
            int b = 0;
            b = 42 / a;
            int c[] = {1};
            c[42] = 99;
        } catch(ArithmetricException e){
            System.out.println( "División por cero. " + e );
        } catch( ArrayIndexOutOfBoundsException e ){
            System.out.println( "Indice fuera de limites. " + e );
        }
        System.out.println( "Después del bloque try/catch. " );
    }
}
```

Practica4Ejercicio06.java

Este programa provoca una división por cero si se ejecuta sin parámetros en la línea de órdenes, ya que **a** es igual a cero. No se producirá este error si se pasa un argumento en la línea de órdenes, ya que **a** tendría un valor mayor que cero. Sin embargo, se producirá una excepción **ArrayIndexOutOfBoundsException**, ya que la matriz de enteros **c** tiene longitud 1 y el programa intenta asignar un valor a **c[42]**.

Pruebe el ejemplo de la siguiente forma y observe la salida generada.

```
make -f makeP4 ejecuta06a
make -f makeP4 ejecuta06b
```

Cuando se utilizan múltiples sentencias **catch** es importante recordar que las subclases de excepción deben estar antes que la superclase. Esto es así porque la sentencia **catch** que utiliza una superclase capturará las excepciones de sus subclases y, por tanto, éstas no se ejecutarán si están después de la superclase. Además, en Java se producirá un error de código no alcanzable.

```
package fp2.poo.practica4;

/**
 * Este programa contiene un error
 * Una subclase debe estar antes que su superclase
 * en una serie de sentencias catch. Si no es así,
 * existirá código que no es alcanzable y se producirá
 * un error en tiempo de compilación.
 */
public class Practica4Ejercicio07 {
    public static void main (String args[]) {
        try {
            int a = 0;
            int b = 42/a;
            System.out.println("Este código nunca se ejecuta");
        } catch ( Exception e ) {
            System.out.println("Sentencia catch para cualquier excepción " + e);
        } catch ( ArithmeticException e ){ //ERROR
            /*
             * Esta sentencia catch nunca se ejecutará ya que la
             * ArithmeticException es una subclase de Exception
             */
            System.out.println( "Esto nunca se alcanza " );
        }
    }
}
```

Practica4Ejercicio07.java

Si intenta compilar este programa se producirá un mensaje de error indicando que la segunda sentencia **catch** no es alcanzable. Como **ArithmeticeException** es una subclase de **Exception**, la primera sentencia **catch** gestionará todos los errores de **Exception**, incluidos los de **ArithmeticeException**. Esto significa que la segunda sentencia **catch** nunca se ejecutará.

Compile el ejemplo de la siguiente forma:

```
make -f makeP4 compila07
```

Para eliminar este problema, es necesario invertir el orden de las sentencias **catch**.

Arregle el ejemplo de la forma indicada para que no presente el problema mencionado.

Sentencias try anidadas

Las sentencias **try** pueden estar anidadas, es decir, una sentencia **try** puede estar dentro del bloque de otra sentencia **try**.

Si en un bloque try se produce una excepción, se busca entre las sentencias catch correspondientes a este try, aquella que trate la excepción que se ha producido. En caso de no encontrarla, se busca en el bloque try inmediatamente superior. Esto continúa hasta que se encuentra la sentencia **catch** adecuada o se agotan las sentencias **try** anidadas. Si no se encuentra la sentencia **catch** buscada, el intérprete de Java gestionará esta excepción.

Este es un ejemplo que utiliza sentencias **try** anidadas.

```
package fp2.poo.practica4;

class Practica4Ejercicio08 {
    public static void main (String args[]) {
        try {
            int a = args.length;
            /*
             * Si no hay ningún argumento en la línea de comandos
             * la siguiente sentencia genera un excepción. Div por 0
             */
            int b = 42/a;
            System.out.println( "a = " + a );

            try {
                /*
                 * bloque try anidado.
                 *
                 * Si se utiliza un argumento en la línea de comandos,
                 * la siguiente sentencia genera una excepc. Div. 0
                 */
                if (a == 1)
                    a = a / (a - a);
                /*
                 * Si se le pasan dos argumento en Linea de Comandos
                 */
                if (a == 2) {
                    int c[] = { 1 };
                    c[42] = 99; //genera excepc. fuera de limites
                }
            } catch ( ArrayIndexOutOfBoundsException e ) {
                System.out.println("Índice fuera de límites. " + e);
            }
            } catch ( ArithmeticException e ){
                System.out.println( "División por cero: " + e );
            }
        }
    }
```

Practica4Ejercicio08.java

Ejecute el programa anterior sin ningún argumento (**ejecuta08a**), con uno (**ejecuta08b**) y con dos argumentos (**ejecuta08c**) en línea de comandos:

```
make -f makeP4 ejecuta08a
make -f makeP4 ejecuta08b
make -f makeP4 ejecuta08c
```

Como se puede ver, este programa anida un bloque **try** dentro de otro. El programa funciona de la siguiente manera:

1. Cuando se ejecuta el programa sin argumentos en la línea de órdenes, se genera una excepción de división por cero en el bloque **try** externo.
2. La ejecución del programa con un argumento en la línea de órdenes genera una excepción de división por cero en el bloque **try** interno. Como el bloque interno no captura esta excepción, se pasa al bloque externo, donde es gestionada.
3. Si se ejecuta el programa con dos argumentos, se produce una excepción en el bloque **try** interno al sobrepasar los límites del tamaño de la matriz.

Nota: Las sentencias **try** anidadas pueden aparecer de formas menos obvias que la anterior cuando están involucradas llamadas a métodos. Por ejemplo, dentro de un bloque **try** puede haber una llamada a un método y dentro de ese método puede haber otra sentencia **try**. En este caso, la sentencia **try** que se encuentra dentro del método está anidada con el bloque **try** exterior que llama al método.

Lanzamiento de una excepción

Hasta ahora, sólo se han mencionado las excepciones generadas por el intérprete de Java. Sin embargo, utilizando la sentencia **throw** es posible hacer que el programa lance una excepción de manera explícita. La forma general de la sentencia **throw** es la siguiente:

```
throw Instanciathrowable;
```

Aquí, *Instanciathrowable* tiene que ser un objeto de tipo **Throwable** o una subclase de **Throwable**. Los tipos simples, como **int** o **char**, y las clases que no son **Throwable**, como **String** y **Object** no se pueden utilizar como excepciones. Hay dos formas de obtener un objeto **Throwable**:

1. Utilizando el parámetro de la cláusula **catch**,
2. Creando un objeto del tipo **Throwable** con el operador **new**.

El flujo de la ejecución se detiene inmediatamente después de la sentencia **throw** y cualquier sentencia posterior no se ejecuta. Se inspecciona el bloque **try** más cercano que la engloba para ver si tiene una sentencia **catch** cuyo tipo coincide con el de la excepción. Si la encuentra, el control se transfiere a esa sentencia. Si no, se inspecciona el siguiente bloque **try** que la engloba y así sucesivamente. Si no encuentra ninguna sentencia **catch** cuyo tipo coincide con el de la excepción, entonces el gestor de excepciones por defecto detiene el programa e imprime el trazado de la pila.

A continuación vemos un ejemplo que crea y lanza una excepción. El gestor que captura la excepción la vuelve a lanzar al gestor más externo:

```

package fp2.poo.practica4;

class Practica4Ejercicio09 {

    /*
     * Las sentencias try pueden estar implícitamente anidadas
     * a través de llamadas a métodos
     */
    static void metodo () {
        try {
            throw new NullPointerException( "demo" );
        } catch ( NullPointerException e ) {
            System.out.println( "Captura dentro de demo" + e );
            /*
             * Vuelve a lanzar la excepción capturada
             */
            throw e;
        }
    }

    public static void main (String args[]){
        try {
            metodo();
        } catch ( NullPointerException e ){
            System.out.println("Nueva captura: " + e );
        }
    }
}

```

Practica4Ejercicio09.java

Este programa tiene dos oportunidades de tratar el mismo error. Primero, **main()** establece un contexto de excepción y después llama a **metodo()**. El método **metodo()** establece después otro contexto de gestión de excepciones y lanza inmediatamente una nueva instancia de **NullPointerException**, que es capturada en la sentencia **catch** de la siguiente línea. Entonces se vuelve a lanzar la excepción.

El programa también muestra cómo crear un objeto de las excepciones estándares de Java. Preste especial atención a la línea siguiente:

```
throw new NullPointerException("demo");
```

Aquí, el operador **new** se utiliza para construir una instancia de **NullPointerException**. Todas las excepciones que están en el núcleo de Java tienen dos constructores:

1. Uno sin parámetros y
2. Otro que tiene un parámetro de tipo cadena.

Cuando se utiliza la segunda forma, el argumento especifica una cadena que describe la excepción. Esta cadena se imprime cuando se utiliza este objeto como argumento de **print()** o **println()**.

Nota: También se puede obtener con una llamada al método **getMessage()**, que está definido en **Throwable**.

Ejecute el programa para probar el ejemplo y observe el resultado.

Declaración de excepciones en métodos

Si un método es capaz de provocar una excepción que no maneja él mismo, debería especificar este comportamiento para que los métodos que lo llaman puedan protegerse frente a esta excepción.

Esto se puede hacer incluyendo una cláusula **throws** en la declaración del método.

Una cláusula **throws** lista los tipos de excepciones que un método puede lanzar. Todas las excepciones que un método puede lanzar deben estar declaradas en una cláusula **throws** (salvo algunas propias de Java), si no están declaradas se producirá un error de compilación.

Esta es la forma general de la declaración de un método que incluye una cláusula **throws**.

```
tipo nombre_del_método(lista_de_parámetros) throws lista_de_excepciones
{
    //cuerpo del método
}
```

Aquí, *lista_de_excepciones* es la lista de excepciones, separadas por comas, que un método puede lanzar.

Este es un ejemplo de un programa incorrecto que intenta lanzar una excepción sin tener código para capturarla. Como el programa no utiliza una cláusula **throws** para declarar esta circunstancia, el programa no compilará.

```
package fp2.poo.practica4;

class Practica4Ejercicio10 {
    static void metodoDemo () {
        System.out.println( "Dentro de metodoDemo " );
        throw new IllegalAccessException( "metodoDemo " );
    }

    public static void main ( String args[] ) {
        metodoDemo ();
    }
}
```

Practica4Ejercicio10.java

Pruebe a compilar el ejemplo de la siguiente forma:

make -f makeP4 compilal0

Para hacer que se compile este ejemplo hay que realizar dos cambios. En primer lugar, es necesario declarar que `metodoDemo()` lanza una excepción `IllegalAccessException`. En segundo lugar, el método `main()` tiene que definir una sentencia `try/catch` que capture esta excepción.

```

package fp2.poo.practica4;

import java.lang.IllegalAccessException;

class Practica4Ejercicio11 {
    static void metodoDemo () throws IllegalAccessException{
        System.out.println("Dentro de metodoDemo");
        throw new IllegalAccessException("metodoDemo");
    }

    public static void main (String args[]){
        try{
            Practica4Ejercicio11.metodoDemo();
        }catch(IllegalAccessException e){
            System.out.println("Captura "+e);
        }
    }
}

```

Practica4Ejercicio11.java

Ejecute el ejemplo de la siguiente forma:

make -f makeP4 ejecutall

Bloque de finalización

Cuando se lanzan excepciones, la ejecución de un método sigue un trayecto no lineal bastante brusco que altera el flujo normal del método. Dependiendo de cómo esté codificado el método, incluso es posible que una excepción provoque que el método termine de forma prematura. Esto puede ser un problema en algunos métodos. Por ejemplo, si un método abre un archivo cuando comienza y lo cierra cuando finaliza, entonces no sería deseable que el mecanismo de gestión de la excepción omita la ejecución del código que cierra el archivo. La palabra clave **finally** está diseñada para resolver este problema.

finally crea un bloque de código que se ejecutará después del bloque **try/catch** y antes del siguiente bloque **try/catch**. El bloque **finally** se ejecutará tanto si se lanza una excepción como si no. Si se lanza una excepción, el bloque **finally** se ejecutará incluso aunque no haya ninguna sentencia **catch** que capture dicha excepción. Siempre que un método vaya a devolver el control al método llamante desde un bloque **try/catch**, mediante una excepción no capturada, o una sentencia **return** explícita, se ejecuta la cláusula **finally** justo antes del final del método. Esto puede resultar útil para cerrar descriptores de archivo y liberar cualquier otro recurso que se hubiese asignado al comienzo de un método con la intención de liberarlo antes de terminar. La cláusula **finally** es opcional. Sin embargo, cada sentencia **try** necesita, al menos, una cláusula **catch** o **finally**.

Este es un ejemplo que muestra tres métodos que terminan de varias maneras, todas ejecutando sus cláusulas **finally**.

```

package fp2.poo.practica4;

import java.lang.IllegalAccessException;

public class Practica4Ejercicio12 {

    /*
     * lanza una excepción fuera del método
     */
    static void metodoA () {
        try {
            System.out.println("Dentro de metodoA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("Sentencia finally metodoA");
        }
    }

    /*
     * ejecuta la sentencia return dentro del bloque try
     */
    static void metodoB () {
        try {
            System.out.println("Dentro de metodoB");
            return;
        } finally {
            System.out.println("Sentencia finally metodoB");
        }
    }

    /*
     * ejecuta un bloque try normalmente
     */
    static void metodoC () {
        try {
            System.out.println("Dentro de metodoC");
        } finally {
            System.out.println("Sentencia finally metodoC");
        }
    }

    public static void main (String args[]){
        try {
            metodoA ();
        } catch ( Exception e ) {
            System.out.println( "Excepción Capturada " + e );
        }
        metodoB ();
        metodoC ();
    }
}

```

Practica4Ejercicio12.java

En este ejemplo, en el método **main**, se invoca al **metodoA()**, al **metodoB()** y al **metodoC()**. El **metodoA()** sale prematuramente de la sentencia **try** lanzando una excepción del tipo **RuntimeException**, y se ejecuta el bloque **finally**. En programa principal se captura la excepción lanzada por el **metodoA()** y continúa su ejecución. El **metodoB()** sale de la sentencia **try** mediante una sentencia **return** y se ejecuta de todas formas la cláusula **finally** antes de que **metodoB()** devuelva el control. En **metodoC()**, la

sentencia **try** se ejecuta normalmente sin que se produzcan errores, y a pesar de ello, también ejecuta el bloque **finally**.

Pruebe el ejemplo y observe el resultado, con la siguiente orden:

```
make -f makeP4 ejecuta12
```

Nota: Si un bloque **finally** está asociado a una sentencia **try**, el bloque **finally** se ejecutará cuando finalice la sentencia **try**.

Excepciones del núcleo de Java

Java define algunas clases de excepciones en su paquete estándar **java.lang**. Algunas de estas clases aparecían en los ejemplos anteriores. Muchas de estas excepciones son subclases del tipo **RuntimeException**. Las excepciones derivadas de **RuntimeException** están disponibles automáticamente en todos los programas Java, ya que el paquete **java.lang** se importa implícitamente. Además, no es necesario incluir estas excepciones en la lista **throws** de ningún método.

En términos de Java, estas excepciones son conocidas con el nombre de *excepciones no comprobadas*, ya que el compilador no comprueba si un método trata o lanza estas excepciones.

Existen otras excepciones definidas en **java.lang** que deben ser incluidas en la lista de excepciones que lanza un método si realmente ese método genera una de estas excepciones y no la trata. Éstas excepciones son conocidas como *excepciones comprobadas*.

Java define otros tipos de excepciones en otros paquetes de su biblioteca de clases.

Excepciones propias

Aunque las excepciones del núcleo de Java tratan la mayoría de los errores comunes, es conveniente poder crear tipos propios de excepción que permitan tratar situaciones específicas en las aplicaciones. Para hacer esto, es necesario definir una subclase de **Exception**, que es una subclase de **Throwable**. Realmente, las excepciones no necesitan implementar nada, simplemente su existencia en el sistema permite utilizarlas como excepciones.

La clase **Exception** no define ningún método, simplemente hereda los métodos definidos por **Throwable**. Por eso, todas las excepciones, incluidas las creadas por nosotros mismos, tienen los métodos definidos por **Throwable**.

El siguiente ejemplo declara una nueva subclase de **Exception** y utiliza esta subclase para indicar una condición de error en un método. Esta clase sobrescribe el método **toString()** (de la clase **Object**), permitiendo que la descripción de la excepción se imprima utilizando **println()**.

```

package fp2.poo.practica4;

import java.lang.Exception;

/**
 * Descripcion: Implementacion de una Excepcion.
 *
 * version 1.0 Mayo 2011
 * Fundamentos de Programacion II
 */
public class Practica4Ejercicio13Exception extends Exception{

    private int detalle;

    public Practica4Ejercicio13Exception(int a){
        detalle = a;
    }

    public String toString(){
        return "Practica4Ejercicio13Exception[" + detalle + "]";
    }
}

```

Practica4Ejercicio13Exception.java

```

package fp2.poo.practica4;

import java.lang.Exception;
import fp2.poo.practica4.Practica4Ejercicio13Exception;

public class Practica4Ejercicio13Main extends Exception{

    static void calcula (int a) throws Practica4Ejercicio13Exception{
        System.out.println("Ejecuta calcula (" + a + ")");
        if ( a > 10 )
            throw new Practica4Ejercicio13Exception(a);
        System.out.println("Finalizacion normal");
    }

    public static void main (String args[]){
        try {
            calcula (1);
            calcula (20);
        } catch ( Practica4Ejercicio13Exception e ){
            System.out.println( "Excepción Capturada " + e );
        }
    }
}

```

Practica4Ejercicio13Main.java

Este ejemplo define una subclase de **Exception** llamada **Practica4Ejercicio13Exception**. Esta subclase es bastante simple, ya que sólo tiene un constructor y un método **toString()** que visualiza el valor de la excepción.

La clase **Practica4Ejercicio13Main**, define un método llamado **calcula()** que lanza un objeto de **Practica4Ejercicio13Exception**.

La excepción es lanzada cuando el parámetro entero de `calcula()` es mayor que 10. El método `main()` establece un gestor de excepciones para **Practica4Ejercicio13Exception**. Se llama a `calcula()` con un valor menor que 10 y con uno mayor que 10 para mostrar los dos caminos que sigue el código.

```
make -f makeP4 ejecuta13
```

Uso de excepciones

La gestión de excepciones proporciona un mecanismo poderoso para controlar programas complejos que tienen muchas características dinámicas durante la ejecución.

Es importante pensar que `try`, `throw` y `catch` son maneras limpias de manejar errores y problemas inesperados en la lógica de un programa. Si es como la mayoría de los programadores, entonces estará acostumbrado a devolver un código de error cuando se produce un fallo en un método. Cuando programe en Java debería terminar con este hábito. Cuando un método puede fallar tiene que lanzar una excepción. Esta es la manera más limpia de gestionar los fallos.

Por último, no debería considerar que las sentencias de gestión de excepciones de Java sean un mecanismo general para realizar ramificaciones, ya que si hace esto, sólo conseguirá enmarañar su código y complicar su mantenimiento.

Uso de excepciones con el jdb

Recuerde que las órdenes `catch` e `ignore` se pueden utilizar en el jdb. La orden `catch` permite indicar al depurador que este debe detener la ejecución del programa en el caso de que se lance la excepción que le indiquemos. La sintaxis de esta orden es `catch excepcion` (incluyendo el paquete al que pertenece la excepción `excepcion`). De forma opuesta, la orden `ignore` nos permite anular el efecto de una orden `catch` previa.

Ejecute la siguiente sentencia

```
jdb -classpath ./bin -sourcepath ./src fp2.poo.practica4.Practica4Ejercicio13Main
```

Capture la excepción que se ha implementado (**fp2.poo.practica4.Practica4Ejercicio13Exception**) con el comando `catch` dentro de jdb, de la siguiente forma:

```
catch fp2.poo.practica4.Practica4Ejercicio13Exception
```

Y ejecute con `run`.

Ejercicio : Modificar la clase **Calculadora** de la práctica 1 para que se limite el rango de los operandos de 0 a 100, en los métodos *sumar*, *restar*, *multiplicar* y *dividir* dos números. Para ello se crea la clase **ValoresFueraDeRangoExcepcion**. En el caso de que los números sobre los que se quiera operar estén fuera de este rango, se debe lanzar esta excepción. Crear un programa principal que utilice la calculadora para hacer varios cálculos y capture la excepción **ValoresFueraDeRangoExcepcion** y muestre un mensaje por pantalla. También se debe capturar la excepción de división por 0 (**ArithmeticException**) en el caso de la división.

Trabajo a entregar.

1. Cree un directorio de nombre su UVUS.
2. Cree en él los directorios **src**, **bin** y **jar**.
3. Cree en **src** los subdirectorios asociados a los paquetes
fp2.poo.practica4XXX.excepcion,
fp2.poo.practica4XXX.interfaz,
fp2.poo.practica4XXX.clase, y
fp2.poo.practica4XXX.principal donde **XXX** es el UVUS del alumno.
4. Cree la clase **AceleracionNoPermitidaExcepcion**, en el paquete **fp2.poo.practica4XXX.excepcion**, con un atributo privado de tipo **String**, dos constructores, uno sin parámetros y otro con un parámetro de tipo **String**.
El constructor sin parámetros le asignará una cadena vacía (“”) al atributo privado.
El constructor con un parámetro usará el parámetro para asignarle un valor al atributo privado.
5. Compile el fichero **AceleracionNoPermitidaExcepcion.java** con el makefile.
6. Cree la clase **DesaceleracionNoPermitidaExcepcion**, en el paquete **fp2.poo.practica4XXX.excepcion**, con un atributo privado de tipo **String**, dos constructores, uno sin parámetros y otro con un parámetro de tipo **String**.
El constructor sin parámetros le asignará una cadena vacía (“”) al atributo privado.
El constructor con un parámetro usará el parámetro para asignarle un valor al atributo privado.
7. Compile el fichero **DesaceleracionNoPermitidaExcepcion.java** con el makefile.
8. Compile con el makefile la siguiente interfaz:

```
package fp2.poo.practica4XXX.interfaz;

import fp2.poo.practica4XXX.excepcion.*;

public interface AccionesEnElCocheInterfaz{
/*
 * Este método incrementa la velocidad actual mediante el incremento del valor proporcionado.
 * @return devuelve la velocidad actual.
 * @throws Este método lanzará la excepción AceleracionNoPermitidaExcepcion cuando reciba valores negativos
 */
public int acelerar (int incremento) throws AceleracionNoPermitidaExcepcion;

/*
 * Este método decrementa la velocidad actual mediante el decrecimiento del valor proporcionado.
 * @return devuelve la velocidad actual.
 * @throws Este método lanzará la excepción DesaceleracionNoPermitidaExcepcion cuando reciba valores
positivos.
 */
public int desacelerar (int decremento) throws DesaceleracionNoPermitidaExcepcion;
}
```

Nota: XXX debe cambiarlo por su uvus. Tenga en cuenta también que debe estar en el directorio apropiado.

9. Implemente la clase **Coche** en el fichero **Coche.java** en el paquete **fp2.poo.practica4XXX.clase**, que implemente la interfaz **AccionesEnElCocheInterfaz**. La clase **Coche** debe contener los siguientes atributos **matricula** y **velocidad**. Debe añadir los métodos **get/set** para los atributos privados correspondientes.
10. Utilice la clase **Main** del fichero **Main.java**, del paquete **fp2.poo.practica4XXX.principal**. **Nota:** XXX debe cambiarlo por su uvus. Tenga en cuenta también que debe estar en el directorio apropiado y que la compilación con el makefile depende de las excepciones.

Fichero: **Main.java**

```
package fp2.poo.practica4XXX.principal;

import fp2.poo.practica4XXX.principal.*;
import fp2.poo.practica4XXX.excepcion.*;
import fp2.poo.practica4XXX.interfaz.*;
import fp2.poo.practica4XXX.clase.*;

class Main{
    public static void main (String args[]){
        Coche coche01 = new Coche ("0000ABC",0);
        int velocidadActual = 0;
        try{
            coche01.acelerar (20);
            velocidadActual = coche01.getVelocidad ();
            System.out.println ("Se invoca al metodo acelerar con parametro 20.");
            System.out.println ("Velocidad Actual: " + velocidadActual );

            System.out.println ("Se invoca al metodo acelerar con parametro -20.");
            coche01.desacelerar (-20);
            velocidadActual = coche01.getVelocidad ();
            System.out.println ("Velocidad Actual: " + velocidadActual );
        }catch (AceleracionNoPermitidaExcepcion e){
            System.out.println("No debe ejecutarse este codigo ");
        }catch (DesaceleracionNoPermitidaExcepcion e){
            System.out.println("No debe ejecutarse este codigo ");
        }

        try{
            coche01.acelerar (-20);
        }catch ( AceleracionNoPermitidaExcepcion e){
            System.out.println("Se ha lanzado la excepcion: "+ e);
        }
        try{
            coche01.desacelerar (20);
        }catch ( DesaceleracionNoPermitidaExcepcion e){
            System.out.println("Se ha lanzado la excepcion: "+ e);
        }
    }
}
```

11. Compile todas las clases y genere el fichero **jar** con el fichero makefile, usando la estructura de directorios utilizada anteriormente (**src, bin y jar**).

El resultado de la ejecución de programa se muestra a continuación:

```
$ java -cp ./bin fp2.poo.practica4XXX.principal.Main
Se invoca al metodo acelerar con parametro 20.
Velocidad Actual: 20
Se invoca al metodo acelerar con parametro -20.
Velocidad Actual: 0
Se ha lanzado la excepcion: AceleracionNoPermitidaExcepcion
Se ha lanzado la excepcion: DesaceleracionNoPermitidaExcepcion
```



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 es 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 indiceIni, int numChars)
```

Aquí, **indiceIni** 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','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 llenen 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 código nunca se ejecuta");  
        } catch (NullPointerException e){  
            System.out.println("invocación incorrecta del método 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.

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

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
    NumeroDeCuentaIncorrectaExpcion;
```

```
}
```

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.pfpooXXX**) 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 proporcionado 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.pfpooXXX`) 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



PRÁCTICA 6: Polimorfismo

OBJETIVO

La programación orientada a objetos es un paradigma de programación que usa objetos y sus interacciones para diseñar las aplicaciones. Se basa en varias técnicas, que incluyen abstracción, encapsulación, polimorfismo y herencia. Esta práctica se centra en el polimorfismo y en la forma de aplicarlo dentro del lenguaje de programación Java. Esta práctica también presenta la herramienta javadoc, que genera documentación HTML a partir de los archivos fuente (.java) de Java.

POLIMORFISMO

El polimorfismo es la habilidad de los objetos de responder con distintos comportamientos ante un mismo mensaje. En Java una forma de utilizar el polimorfismo es mediante la sobrecarga de métodos.

Sobrecarga de método

La sobrecarga es la posibilidad de tener dos o más funciones o métodos con el mismo nombre pero funcionalidad diferente. Es decir, dos o más operaciones con el mismo nombre realizan acciones diferentes. El compilador usará una u otra dependiendo de los parámetros usados.

Java permite definir dos o más métodos dentro de la misma clase que tengan el mismo nombre, pero con sus listas de parámetros distintas. Cuando ocurre esto, se dice que los métodos están **sobrecargados** y a este proceso se le denomina **sobrecarga de método**.

Cuando se invoca a un método sobrecargado, Java utiliza el tipo y/o el número de argumentos como guía para determinar la versión del método sobrecargado que realmente debe llamar. Por eso, los métodos sobrecargados deben diferenciarse en el tipo y/o en el número de parámetros. Aunque los métodos sobrecargados puedan devolver diferentes tipos de valores, el tipo devuelto por sí solo, es insuficiente para distinguir dos versiones del método. Cuando Java encuentra una llamada a un método sobrecargado, simplemente ejecuta la versión del método cuyos parámetros coinciden con los argumentos utilizados en la llamada al método.

Veamos un ejemplo de sobrecarga de métodos:

```
package fp2.poo.practica6;

public class Practica6Ejercicio01 {

    public void metodoSobrecargado() {
        System.out.println("Sin parametros");
    }

    public void metodoSobrecargado(int a) {
        System.out.println("Con un parametro entero: " + a);
    }

    public void metodoSobrecargado(int a, int b) {
        System.out.println("Con dos parametros a y b: practica6 " + a + " " + b );
    }

    public double metodoSobrecargado(double a) {
        System.out.println("Con un parametro double: " + a);
        return a * a;
    }

    //public int metodoSobrecargado(int a) {
    //    System.out.println("Con un parametro entero: " + a);
    //}
}

class Main {
    public static void main( String args[ ] ) {
        Practica6Ejercicio01 obj = new Practica6Ejercicio01 ();
        double result = 0.0d;
        obj.metodoSobrecargado();
        obj.metodoSobrecargado(10);
        obj.metodoSobrecargado(10, 20);
        result = obj.metodoSobrecargado(123.2d);
        System.out.println(" valor devuelto al invocar a obj.metodoSobrecargado(123.2d): "
                           + result );
    }
}
```

Practica6Ejercicio01.java

Descargue y descomprima el código de la plataforma para realizar la práctica 6.

Observe que el fichero `Practica6Ejercicio01.java`, contiene dos clases `Practica6Ejercicio01` y `Main`. Una de ellas es pública y otra visible a nivel de paquete. Este será el formato general para las pruebas en esta práctica, aunque se recomienda utilizar un fichero por clase de forma habitual. Y recuerde que la compilación en este caso genera dos archivos `.class`, uno por clase.

Use el *makefile* proporcionado y compile y ejecute el programa mediante el siguiente comando:

```
make -f makeP6 prueba01
```

Cambie el nivel de acceso de la clase `Main` a nivel `public` y compílelo de nuevo con el comando anterior. Observe que produce un error en compilación debido a que aparecen dos clases públicas en el mismo fichero.

Obsérvese que `metodoSobrecargado()` ha sido sobrecargado cuatro veces (y además una versión con comentarios). La primera versión no tiene parámetros; la segunda tiene un parámetro entero; la tercera tiene dos parámetros enteros y la cuarta tiene un parámetro `double`. El hecho

de que la cuarta versión de **metodoSobrecargado()** devuelva un valor de tipo **double** no está relacionado con la sobrecarga, ya que los tipos devueltos no juegan ningún papel en la resolución de la sobrecarga.

Quite los comentarios al quinto método sobrecargado y ejecute de nuevo el comando anterior e interprete el resultado.

Promoción automática de tipo en la sobrecarga de método

Cuando se llama a un método sobrecargado, Java busca una versión del método cuyos parámetros coincidan con los argumentos utilizados en la llamada al método. Sin embargo, esta coincidencia no tiene por qué ser siempre exacta. En algunos casos, las conversiones de tipo automáticas de Java pueden jugar un papel importante en la resolución de la sobrecarga. Por ejemplo, consideremos el siguiente programa.

```
package fp2.poo.practica6;

public class Practica6Ejercicio02 {

    public void metodoSobrecargado() {
        System.out.println("Sin parámetros");
    }

    //public void metodoSobrecargado(int a) {
    //    System.out.println("Con un parametro entero: "+a);
    //}

    public double metodoSobrecargado(double a) {
        System.out.println("Con un parametro double: "+a);
        return a * a;
    }
}

class Main {
    public static void main( String args[ ] ) {
        Practica6Ejercicio02 ob = new Practica6Ejercicio02();
        int i = 88;
        ob.metodoSobrecargado();
        ob.metodoSobrecargado(i);      //Esto llama a metodoSobrecargado (double)
        ob.metodoSobrecargado(123.2); //Esto llama a metodoSobrecargado (double)
    }
}
```

Practica6Ejercicio02.java

Compile y ejecute el programa mediante el siguiente comando.

```
make -f makeP6 prueba02
```

Obsérvese que en esta versión no se define el método **metodoSobrecargado(int)**. Cuando se llama a **metodoSobrecargado()** con un argumento entero, no se encuentra ningún método cuyos parámetros coincidan exactamente con el argumento. Sin embargo, Java puede convertir automáticamente un entero en un **double** y esta conversión puede ser utilizada para resolver la llamada al método. Por eso, cuando no encuentra el método **metodoSobrecargado(int)**, Java convierte la variable **i** a tipo **double** y llama a **metodoSobrecargado(double)**. Por supuesto, si hubiese estado definido el método

metodoSobrecargado(int), Java lo habría llamado. Java emplea su conversión de tipo automática sólo si no existe una coincidencia exacta entre parámetros y argumentos.

Quite los comentarios del método **metodoSobrecargado(int)**, vuelva a ejecutar el comando anterior, y observe el resultado.

Ejercicio:

En el siguiente ejemplo se prueban distintas promociones de tipos en la llamada a métodos sobrecargados.

Cada tipo de dato simple contiene su implementación de método sobrecargo. Se trata de ver a qué tipo promociona si no existe la versión de su método.

Sustituya XXX por cada uno de los tipos para los que está implementada la sobrecarga de método. Una vez que haya sustituido XXX por un tipo concreto (por ejemplo por byte), comente el método que contiene un parámetro del tipo que haya elegido (Si XXX lo ha sustituido por byte, comente las tres líneas del método public void metodoSobrecargado(byte arg)).

```
package fp2.poo.practica6;

public class Practica6Ejercicio03 {

    public void metodoSobrecargado(byte arg) {
        System.out.println("Metodo: metodoSobrecargado(byte obj)");
    }

    public void metodoSobrecargado(short arg) {
        System.out.println("Metodo: metodoSobrecargado(short obj)");
    }

    public void metodoSobrecargado(int arg) {
        System.out.println("Metodo: metodoSobrecargado(boolean obj)");
    }

    public void metodoSobrecargado(char arg) {
        System.out.println("Metodo: metodoSobrecargado(char obj)");
    }

    public void metodoSobrecargado(long arg) {
        System.out.println("Metodo: metodoSobrecargado(long obj)");
    }

    public void metodoSobrecargado(float arg) {
        System.out.println("Metodo: metodoSobrecargado(float obj)");
    }

    public void metodoSobrecargado(double arg) {
        System.out.println("Metodo: metodoSobrecargado(double obj)");
    }

    public void metodoSobrecargado(boolean arg) {
        System.out.println("Metodo: metodoSobrecargado(boolean obj)");
    }
}

class Main {
    public static void main( String args[ ] ) {
        DemoSobrecarga ob = new DemoSobrecarga ();
        /*
         * Ponga el tipo a la variable arg y elimine el metodo sobrecargado
         * al que se le asociaria.
         */
        XXX arg = 88;
        ob.metodoSobrecargado( arg );
    }
}
```

Practica6Ejercicio03.java

Este ejemplo necesita volver a compilarlo cada vez que se realiza la eliminación de un método para probar la promoción del tipo de dato. Pruébelo para todos los tipos de datos simple mediante:

```
make -f makeP6 prueba03
```

Sobrecarga de constructores

Además de sobrecargar métodos normales, también se pueden sobrecargar los constructores. De hecho, en la mayoría de las clases que se implementan en el mundo real, la sobrecarga de los constructores es la norma y no la excepción. Vamos a aplicarlo a la clase Saldo() desarrollada en la práctica anterior:

```
package fp2.poo.practica6;

public class Saldo {
    Double saldo;

    public Saldo() {
        saldo = 0.0d;
    }

    public Saldo(Double d) {
        saldo = d;
    }

    public Saldo(double d) {
        this.saldo = new Double(d);
    }

    public double getSaldo() {
        return this.saldo.doubleValue();
    }

    public void setSaldo(double d) {
        this.saldo = new Double(d);
    }
}
```

Saldo.java

Observe los tres constructores que se proporcionan en la clase Saldo.

```
package fp2.poo.practica6;

public class Practica6Ejercicio04 {
    public static void main (String args[]){
        Saldo obj1 = new Saldo();
        Saldo obj2 = new Saldo(new Double(5e+10));
        Saldo obj3 = new Saldo(20.5d);
        System.out.println("obj1.getSaldo() = " + obj1.getSaldo());
        System.out.println("obj2.getSaldo() = " + obj2.getSaldo());
        System.out.println("obj3.getSaldo() = " + obj3.getSaldo());
    }
}
```

Practica6Ejercicio04.java

Compile y ejecute el programa con el siguiente comando:

```
make -f makeP6 prueba04
```

Objetos como parámetros

Hasta ahora sólo se han utilizado tipos simples en los parámetros de los métodos. Sin embargo, es correcto y habitual pasar objetos a los métodos. Consideremos la siguiente versión de la clase **Saldo** en la que se ha incluido un nuevo constructor y el método **igual**.

```
package fp2.poo.practica6;

public class Saldo  {

    Double saldo;

    public Saldo() {
        saldo = 0.0d;
    }

    /*
     * Ejemplo de objeto como parametro
     */
    public Saldo(Saldo obj) {
        this.saldo = ((obj == null) ? null : obj.getSaldo());
    }

    public Saldo(Double d) {
        saldo = d;
    }

    public Saldo(double d) {
        this.saldo = new Double(d);
    }

    public double getSaldo() {
        return this.saldo.doubleValue();
    }

    public void setSaldo(double d) {
        this.saldo = new Double(d);
    }

    /*
     * Ejemplo de objeto como parametro
     */
    public boolean igual (Saldo obj) {
        boolean resultado = false;

        if ( (obj != null) && (obj.getSaldo() == this.getSaldo()) ) {
            resultado= true;
        } else {
            resultado= false;
        }
        return resultado;
    }

}
```

Saldo.java

Como se puede ver, el método **igual()** dentro de **Saldo** compara dos objetos para ver si son iguales y devuelve el resultado. Es decir, compara el objeto que invoca al método con uno que se pasa como parámetro. Si tiene los mismos valores, entonces el método devuelve **true**. En caso contrario, devuelve **false**. Observe que el parámetro **obj** de **igual()** es de tipo **Saldo**.

En el siguiente fichero se proporciona una clase en la que se usan los métodos añadidos a la clase **Saldo**.

```
package fp2.poo.practica6;

public class Practica6Ejercicio05 {
    public static void main (String args[]){
        Saldo saldo1 = new Saldo( 6000.33d );
        Saldo saldo2 = new Saldo(saldo1);
        System.out.println("saldo1 .getSaldo() = " + saldo1.getSaldo());
        System.out.println("saldo2 .getSaldo() = " + saldo2.getSaldo());
        System.out.println("saldo1 == saldo2 : " + saldo1.igual(saldo2));
    }
}
```

Practica6Ejercicio05.java

Compile y ejecute el programa con el siguiente comando:

```
make -f makeP6 prueba05
```

Paso de argumentos

En general, en los lenguajes de programación hay dos formas de pasar un argumento a un método.

1. La primera forma es el **paso de parámetros por valor**. En este caso se copia el **valor** del argumento en el parámetro formal del método. Los cambios que se realizan sobre el parámetro formal del método no tienen efecto sobre el argumento utilizado en la llamada. En Java cuando se pasa un tipo simple a un método, se pasa por valor.
2. La segunda forma es el **paso de parámetros por referencia**. En este caso, el parámetro formal recibe la referencia del argumento utilizado en la llamada. Dentro del método, esta referencia se utiliza para acceder al argumento real especificado en la llamada. Esto significa que los cambios realizados al parámetro afectarán al argumento utilizado en la llamada del método. En Java cuando se pasa un objeto a un método, se pasa por referencia.

Devolución de objetos

Un método puede devolver cualquier tipo de dato, incluyendo los tipos de clases definidos por el programador. Se le ha añadido a la clase **Saldo** un método llamado **crearCopia()** que devuelve un nuevo objeto de tipo **Saldo**.

En el siguiente ejemplo se proporciona una clase en la que se usa este método de la clase **Saldo**.

```
package fp2.poo.practica6;

public class Practica6Ejercicio06 {
    public static void main (String args[]){
        Saldo saldo1 = new Saldo( 6000.33d );
        Saldo saldo2 = saldo1.crearCopia();
        System.out.println("saldo1 .getSaldo() = " + saldo1.getSaldo());
        System.out.println("saldo2 .getSaldo() = " + saldo2.getSaldo());
        System.out.println("saldo1 == saldo2 : " + saldo1.igual(saldo2));
    }
}
```

Practica6Ejercicio06.java

Nótese que la clase Saldo.java, incorpora además de lo mostrado anteriormente el siguiente método:

```
/*
 * Ejemplo de devolución de objetos.
 */
public Saldo crearCopia() {
    return new Saldo(this);
}
```

Método crearCopia() de la clase Saldo.java

Compile y ejecute el programa con el siguiente comando:

```
make -f makeP6 prueba06
```

El programa anterior presenta otro aspecto importante. Como todos los objetos se crean dinámicamente utilizando el operador **new**, no es necesario preocuparse de su existencia una vez que el método en el que fue creado termine, ya que el objeto seguirá existiendo mientras haya una referencia a él en alguna parte del programa. Cuando no existan más referencias a ese objeto, entonces será eliminado por el sistema de recogida de basura.

Argumentos en la línea de órdenes (comandos)

Algunas veces es necesario pasar información a un programa cuando éste se ejecuta. Esto se consigue mediante el paso de *argumentos en la línea de órdenes* a **main()**. Los argumentos de la línea de órdenes es la información que directamente sigue al nombre del programa en la línea de órdenes cuando se ejecuta el programa. Acceder a los argumentos de la línea de órdenes dentro de un programa Java es bastante fácil, ya que son almacenados como cadenas en la matriz de **String** que se pasa a **main()**. Se puede obtener el número de argumentos en la línea de comandos con el atributo **length** del array. En Java, se conoce siempre el nombre de la aplicación ya que es el nombre de la clase en la cual el método principal está definido. Por esto el sistema de ejecución de Java no pasa el nombre de la clase que se invoca al método principal.

El siguiente programa muestra todos los argumentos que recibe de la línea de órdenes.

```
package fp2.poo.practica6;

public class Practica6Ejercicio07 {
    public static void main (String args[]){
        for ( int i = 0; i < args.length ; i++) {
            System.out.println("args [" + i + "] : " + args[i]);
        }
    }
}
```

Practica6Ejercicio07.java

Compile y ejecute el programa con el siguiente comando:

```
make -f makeP6 prueba07
```

Ejercicios.

1. Descomprima el fichero **CodigoDeLaParteFinalDeLaPractica6.zip**. Proporcionado en esta práctica. Dada la interfaz TitularInterfaz.java del paquete **fp2.poo.utilidades**, cuyo código aparece a continuación:

```
package fp2.poo.utilidades;

import fp2.poo.pfpooXXX.Dni;

/**
 * Descripcion: Esta es una clase que representa un usuario de
 * una cuenta bancaria.
 *
 * @version version 1.0 Mayo 2011
 * @author Fundamentos de Programacion II
 */
public interface TitularInterfaz {

    /**
     * Descripcion: Metodo de configuracion del atributo nombre.
     */
    public void setNombre( String nombre );

    /**
     * Descripcion: Metodo getter de nombre.
     */
    public String getNombre( );

    /**
     * Descripcion: Metodo de configuracion del atributo
     * relacionado con el primer apellido.
     */
    public void setPrimerApellido( String primerApellido );

    /**
     * Descripcion: Metodo getter del primer apellido.
     */
    public String getPrimerApellido( );

    /**
     * Descripcion: Metodo de configuracion del atributo
     * relacionado con el segundo apellido.
     */
    public void setSegundoApellido( String segundoApellido );

    /**
     * Descripcion: Metodo getter del segundo apellido.
     */
    public String getSegundoApellido( );

    /**
     * Descripcion: Metodo getter del dni.
     */
    public void setDni(Dni obj );

    /**
     * Descripcion: Metodo getter del dni.
     */
    public Dni getDni( );

    /**
     * Descripcion: Metodo de configuracion del atributo domicilio.
     */
    public void setDomicilio( String domicilio );

    /**
     * Descripcion: Metodo getter del domicilio.
     */
    public String getDomicilio( );
}
```

TitularInterfaz.java

Implemente la clase **Titular** en el paquete **fp2.poo.pfpooXXX**, siendo **xxx** el **login** del alumno proporcionado por el Centro de Cálculo (CDC).

La clase **Titular** mantiene la información relacionada con el titular de una cuenta bancaria. Deberá tener los atributos privados necesarios para almacenar dicha información.

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

```
package fp2.poo.utilidades;

import fp2.poo.pfpooXXX.Titular;
import fp2.poo.pfpooXXX.NumeroDeCuenta;
import fp2.poo.pfpooXXX.Saldo;

/**
 * Descripcion: Esta es una clase que representa una cuenta bancaria.
 * Mantiene una asociacion entre un usuario (de tipo Usuario),
 * su saldo (de tipo Saldo) y numero de cuenta
 * (de tipo NumeroDeCuenta).
 *
 * @version version 1.0 Mayo 2011
 * @author Fundamentos de Programacion II
 */
public interface CuentaBancariaInterfaz {

    /**
     * Descripcion: Configura el saldo de una cuenta.
     */
    public void setSaldo (Saldo saldo);

    /**
     * Descripcion: Devuelve el saldo de una cuenta.
     */
    public Saldo getSaldo();

    /**
     * Descripcion: Configura el numero de cuenta de una cuenta.
     */
    public void setNumeroDeCuenta (NumeroDeCuenta numeroDeCuenta);

    /**
     * Descripcion: Devuelve el numero de cuenta de una cuenta.
     */
    public NumeroDeCuenta getNumeroDeCuenta();

    /**
     * Descripcion: Configura el titular de cuenta de una cuenta.
     */
    public void setTitular (Titular titular);

    /**
     * Descripcion: Devuelve el titular de cuenta de una cuenta.
     */
    public Titular getTitular();
}
```

NumeroDeCuentaInterfaz.java

Implemente la clase **CuentaBancaria** en el paquete **fp2.poo.pfpooXXX**, siendo **xxx** el **login** del alumno proporcionado por el Centro de Cálculo (CDC).

La clase **CuentaBancaria** mantiene la información relacionada con una cuenta bancaria.

GESTION DE DOCUMENTACIÓN (**javadoc**)

Los comentarios de documentación, conocidos normalmente como *comentarios doc* o *comentarios javadoc*, permiten asociar directamente a nuestro código documentación de referencia para programadores utilizando el formato HTML.

Los comentarios de documentación se diseñan generalmente para introducir documentación de referencia en las clases con el nivel de detalle que la mayoría de los programadores necesita para la utilización de estas clases.

Un procedimiento para generar documentación a partir de los comentarios de documentación es usar el programa **javadoc**.

El programa **javadoc** se encarga de generar de manera automática la documentación correspondiente al código fuente que se ha creado. Para ello el programador debe añadir ciertos comentarios en su código fuente (siguiendo unas reglas) de forma que el programa **javadoc** pueda reconocerlas y así crear la documentación correspondiente. La sintaxis de este programa es la siguiente:

```
javadoc [opciones] [paquetes] [ficheros_java]
```

donde:

- [opciones] modifican el comportamiento de la herramienta **javadoc**. Algunas de las opciones más frecuentes son las siguientes:

-public	Genera la documentación correspondiente a las clases y miembros con modificadores <code>public</code> presentes en el código fuente a documentar.
-protected	Genera la documentación correspondiente a las clases y miembros con modificadores <code>public</code> y <code>protected</code> presentes en el código fuente a documentar. Esta es la opción por defecto.
-package	Genera la documentación correspondiente a las clases y miembros con modificadores <code>public</code> , <code>protected</code> y <code>package</code> presentes en el código fuente a documentar.
-private	Genera la documentación correspondiente a todas clases y miembros presentes en el código fuente a documentar.
-d directorio	Opción encargada de indicar a javadoc el directorio destino de la documentación a generar.
-sourcepath ruta	Opción encargada de indicar a javadoc la ruta donde se encuentran los paquetes que se indican en el comando. Cada una de las rutas se indicarán separadas por <code>:</code> .
-charset	Controla la codificación de los ficheros generados. En esta práctica se utilizará la codificación “UTF-8”.

- La opción paquetes indica los paquetes de los que se desea generar la documentación. La ubicación de los paquetes vendrá dada por el valor de la opción `-sourcepath` que se indique previamente.
- La opción `ficheros_java` indica los ficheros con el código fuente (`ficheros.java`) de los que se desea generar la documentación. Se tendrá que especificar la ruta correspondiente, no utilizándose en este caso el valor de la opción `-sourcepath`.

Ejercicio 1:

Se proporciona la siguiente interfaz:

```
package fp2.poo.practica6.javaDoc;

public interface SaldoInterfaz{
    public double getSaldo();
    public Double getSaldoDouble();
    public void setSaldo(Double d);
    public void setSaldo(double d);
}
```

SaldoInterfaz.java

Ejecute la siguiente orden:

```
make -f makeJavadoc prueba01
```

Esta orden genera mediante el programa `javadoc` documentación asociada a la interfaz `SaldoInterfaz.java`, en el directorio `doc`. Utilice el navegador para visualizar el fichero `index.html` que se ha generado en el directorio `doc`, y navegue por la información generada por la herramienta.

La anatomía de un comentario de documentación

Los comentarios de documentación comienzan con los tres caracteres `/**` y se extienden hasta el siguiente `*/`. Todos los comentarios de documentación describen el identificador cuya declaración sigue inmediatamente después. Los caracteres `*` iniciales se ignoran en las líneas de estos comentarios, y también los espacios en blanco precediendo al `*` inicial. La primera frase del comentario es el resumen del identificador, donde “frase” significa todo el texto hasta el primer punto seguido por un espacio en blanco. Consideremos el siguiente comentario de documentación:

```
/**
 * Devuelve el saldo. El valor devuelto es de
 * tipo double.
 */
public double getSaldo();
```

El resumen del método `getSaldo` será “**Devuelve el saldo**”. La primera frase de un comentario de documentación debe ser un buen resumen.

Reconvierte los comentarios dados en `SaldoInterfaz.java` en comentarios al estilo **javadoc** (cambiar `/*` por `/**`) y genere de nuevo la documentación mediante el comando dado anteriormente.

A menudo se insertan etiquetas de HTML en los comentarios de documentación, que actúan como enlaces de referencias cruzadas con otra documentación. Podemos utilizar casi cualquier etiqueta de HTML, excepto las etiquetas de cabecera `<h1>`, `<h2>...`, que están reservadas para su uso en la documentación generada. Para insertar los caracteres `<`, `>`, o `&` debe utilizarse `<`, `>` o `&` respectivamente. Si hay que poner un carácter `@` al principio de una línea, debe utilizarse `@`, si no, se supone que es el comienzo de una etiqueta de un comentario de documentación.

Sólo se procesan los comentarios de documentación inmediatamente anteriores a una clase, interfaz, método o atributo. Si hay algo diferente de espacios en blanco o comentarios entre un comentario de este tipo y lo que describe, el comentario se ignora. Por ejemplo, si ponemos un comentario de documentación al principio de un archivo y existe una sentencia `package` o `import` entre el comentario y una sentencia `class`, el comentario no se utilizará. Los comentarios de documentación se aplican a todos los identificadores que se declaran en una sola sentencia, por tanto se evitan este tipo de declaraciones cuando se van a utilizar estos comentarios.

Si no se proporciona un comentario de documentación para un método heredado, el método “hereda” el comentario. Esto en general es suficiente, especialmente cuando una clase hereda una interfaz. Los métodos de la implementación generalmente no hacen más que lo que la interfaz especifica, o al menos nada que no esté descrito en un comentario de documentación. Es conveniente poner un comentario explícito cuando heredemos comentarios de documentación, para que nadie pueda pensar que olvidamos documentar el método. Por ejemplo, en la clase `Saldo` que implementa la interfaz se puede indicar de la siguiente forma:

```
// Hereda comentarios de documentacion
public double getSaldo() {
    //...
}
```

si un método hereda comentarios de documentación de una superclase y una interfaz, se utiliza el comentario de la interfaz.

La clase `Saldo00` proporcionada a continuación implementa la interfaz `SaldoInterfaz`. Esta clase hereda todos los comentarios de la interfaz que implementa, por tanto se ha indicado en el método de la clase.

```

package fp2.poo.practica6.javaDoc;

import fp2.poo.practica6.javaDoc.SaldoInterfaz;

public class Saldo00 implements SaldoInterfaz {

    Double saldo;

    //Hereda comentarios de documentacion
    public Saldo00(Double d) {
        saldo = d;
    }

    //Hereda comentarios de documentacion
    public Saldo00(double d) {
        this.saldo = new Double(d);
    }

    //Hereda comentarios de documentacion
    public double getSaldo() {
        return this.saldo;
    }

    //Hereda comentarios de documentacion
    public Double getSaldoDouble() {
        return this.saldo;
    }

    //Hereda comentarios de documentacion
    public void setSaldo(Double d) {
        this.saldo = d;
    }

    //Hereda comentarios de documentacion
    public void setSaldo(double d) {
        this.saldo = new Double(d);
    }
}

```

Saldo00.java

Ejecute la siguiente orden

make -f makeJavadoc prueba02

Observe la documentación generada.

Etiquetas

Las etiquetas que se pueden utilizar en los comentarios javadoc son las siguientes:

- **@author** nombre

Esta etiqueta sirve para indicar el autor de la clase. Se pueden especificar tantos párrafos de **@author** como deseemos.

- **@version** datos de la versión (número y fecha)

Esta etiqueta sirve para indicar la versión del software que contiene esta clase.

- **@since** version inicial en la que aparece

La etiqueta **@since** permite indicar una especificación de versión con información de cuándo la entidad etiquetada se añadió al sistema.

Etiquetar la “versión de nacimiento” puede ayudarnos a seguir qué entidades son nuevas, y por tanto necesitan documentación o pruebas más intensas. Una etiqueta **@since**, en una clase o interfaz aplica a todos sus miembros que no tengan su propia etiqueta **@since**.

- **@param** nombre

La etiqueta **@param** documenta un solo parámetro de un método. Si utilizamos etiquetas **@param** hay que emplear una por cada parámetro del método. A la etiqueta **@param** debe seguirle el nombre del parámetro del método que documenta

- **@return** valor devuelto

La etiqueta **@return** documenta el valor de retorno de un método.

- **@throws** y **@exception**

La etiqueta **@throws** documenta una excepción lanzada por un método. Si utilizamos etiquetas **@throws**, debe haber una por cada tipo de excepción que lance el método. Esta lista a menudo es más amplia que sólo las excepciones comprobadas de la cláusula **throws**. Es una buena idea declarar todas las excepciones en la cláusula **throws**, se requieran o no, y esto también es cierto al utilizar etiquetas **@throws**. Por ejemplo, supongamos que nuestro método comprueba sus parámetros para asegurarse de que ninguno de ellos es **null**, y lanza una excepción **NullPointerException** si encuentra un parámetro **null**. Debemos declarar la excepción **NullPointerException** en nuestra cláusula **throws** y en nuestras etiquetas **@throws**. La etiqueta **@exception** es equivalente a **@throws**.

- **@deprecated**

La etiqueta **@deprecated** marca un identificador como desaprobado, es decir, inadecuado para continuar con su uso. El código que utiliza un tipo, constructor o campo desaprobado puede generar un aviso cuando compila. Es conveniente asegurarse de que la entidad desaprobada continúa funcionando de forma que no dañemos código existente que no haya sido actualizado. La desaprobación nos ayuda a animar a los usuarios de nuestro código a actualizarse a la última versión, preservando la integridad del código existente. Los usuarios pueden moverse a las nuevas versiones cuando lo prefieran, en vez de ser forzados a actualizarse tan pronto como lancemos una nueva versión de nuestros tipos. Debemos recomendar a los usuarios que sustituyan los tipos desaprobados. Observe que en la clase **String** aparecen algunos métodos y constructores marcados como *deprecated*, en el siguiente enlace:

<http://download.oracle.com/javase/1.4.2/docs/api/java/lang/String.html>

- **@see**

La etiqueta **@see** crea un enlace de referencia cruzada a otra documentación **javadoc**. Se puede nombrar cualquier identificador, aunque debe calificarse de la siguiente forma **paquete.clase#miembro**. Por ejemplo, generalmente podremos nombrar a un miembro de una clase utilizando su nombre simple. Sin embargo, si el miembro es un método sobrecargado, deberemos especificar a qué sobrecarga del método nos referimos indicando los tipos de los parámetros. Podemos especificar una clase o interfaz que pertenezca al paquete actual utilizando su nombre sin calificar, pero los tipos de otros paquetes se deben especificar con sus nombres completamente calificados. Los miembros de tipos se especifican utilizando # antes de su nombre.

En el siguiente ejemplo se muestra la clase `Saldo01` que implementa la interfaz `SaldoInterfaz`, en ella se muestran varios ejemplos de las etiquetas comentadas anteriormente.

```

package fp2.poo.practica6.javaDoc;

import fp2.poo.practica6.javaDoc.SaldoInterfaz;

/**
 * Clase de ejemplo para mostrar el funcionamiento de javadoc.
 *
 * @author Fundamentos de Programacion II
 * @since 18-Mayo-2011
 * @version 1.0
 */
public class Saldo01 implements SaldoInterfaz {

    Double saldo;

    /**
     * Constructor de Saldo01.
     *
     * @param d de tipo Double
     * @see Saldo01#Saldo01(double)
     */
    public Saldo01(Double d) {
        saldo = d;
    }

    /**
     * Constructor de Saldo01.
     *
     * @param d de tipo double
     * @see Saldo01#Saldo01(Double )
     */
    public Saldo01(double d) {
        this.saldo = new Double(d);
    }

    /**
     * Devuelve el saldo.
     *
     * @return devuelve el saldo como double
     * @see Saldo01#getSaldoDouble()
     */
    public double getSaldo() {
        return this.saldo;
    }

    /**
     * Devuelve el saldo en un objeto Double.
     *
     * @return devuelve el saldo como Double
     * @see Saldo01#getSaldo()
     */
    public Double getSaldoDouble() {
        return this.saldo;
    }

    /**
     * Devuelve el saldo en un objeto Double.
     *
     * @param d de tipo Double.
     * @see Saldo01#getSaldo(double)
     * @throws NullPointerException si el objeto es null.
     */
    public void setSaldo(Double d) throws NullPointerException {
        this.saldo = d;
    }

    /**
     * Devuelve el saldo en un objeto Double.
     *
     * @param d de tipo double.
     * @see Saldo01#getSaldo(Double)
     */
    public void setSaldo(double d) {
        this.saldo = new Double(d);
    }
}

```

Saldo01.java

Ejecute la siguiente orden

make -f makeJavadoc prueba03

La siguiente tabla muestra el ámbito en el que se pueden utilizar las etiquetas `javadoc`:

	Comentarios en Clases	Comentarios en métodos
<code>@author</code>	Si	NO
<code>@param</code>	No	Si
<code>@return</code>	No	Si
<code>@throws</code>	No	Si
<code>@version</code>	Si	No
<code>@see</code>	Si	Si
<code>@since</code>	Si	Si
<code>@deprecated</code>	Si	Si

Ejercicio

Ponga los comentarios `javadoc` a las siguientes clases ya implementadas en las prácticas anteriores, siguiendo el ejemplo proporcionado en esta práctica:

- Saldo.java
- SaldoInterfaz.java
- Dni.java
- DniInterfaz.java
- NumeroDeCuenta.java
- NumeroDeCuentaInterfaz.java
- TitularInterfaz.java
- Titular.java
- CuentaBancariaInterfaz.java
- CuentaBancaria.java

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 `Titular` y `CuentaBancaria`, implementados en la anterior sección Ejercicios. Proporcione un makefile para realizar la compilación y ejecución. Nota: la clase `Titular` usa `Dni`, utilizado en la práctica anterior, y la clase `CuentaBancaria` utiliza las clases `Titular`, `NumeroDeCuenta`, y `Saldo`, y por tanto deben estar definidas y compiladas.
2. Comprima el directorio de nombre su `uvus` en el fichero `uvus.zip` y entréguelo en la actividad correspondiente a la práctica.



PRACTICA 7: HERENCIA

1. OBJETIVO

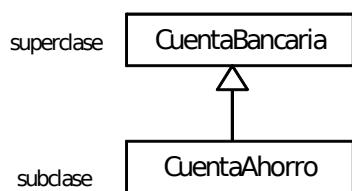
El objetivo de esta práctica es presentar la herencia en el entorno de programación elegido para realizar las prácticas de Fundamentos de Programación II. Descargue el código de la plataforma virtual para realizar la práctica 7. Se proporciona un makefile **makeP7** para la realización de la práctica 7.

2. HERENCIA

La herencia es una de las piedras angulares de la programación orientada a objetos ya que permite la creación de clasificaciones jerárquicas. Gracias a la herencia, se puede definir una clase general que define características comunes a un conjunto de elementos relacionados. Esta clase puede ser heredada por otras clases más específicas, añadiendo cada una de éstas aquellas cosas que son particulares a ella. En la terminología de Java, a la clase general se le llama **superclase**. A las clases más específicas que heredan de la general se les llama **subclases**. Una subclase es una versión especializada de una superclase, que hereda todas las variables de instancia y los métodos definidos por la superclase y que añade sus propios elementos.

Fundamentos

Para heredar una clase, simplemente es necesario incorporar su definición en la definición de otra clase utilizando la palabra clave **extends**. Para entender cómo se realiza esto comencemos con un ejemplo sencillo. El siguiente programa crea una superclase llamada **CuentaBancaria** y una subclase llamada **CuentaAhorro**. Esto lo representamos de forma gráfica de la siguiente forma:



Observe cómo se utiliza **extends** para crear la subclase **CuentaAhorro** de **CuentaBancaria**.

Primero la superclase:

```
package fp2.poo.practica7;

public class CuentaBancaria{

    private double saldo;

    public CuentaBancaria() {
        this.saldo = 0.;
    }

    public CuentaBancaria(double cantidadInicial) {
        System.out.println("Constructor de CuentaBancaria");
        this.saldo = cantidadInicial;
    }

    public double getSaldo(){
        return this.saldo;
    }
    public void setSaldo(double saldo){
        this.saldo = saldo;
    }

    public void retirar(double cantidad) throws NoFondosDisponiblesException {
        if( cantidad > 0 ){
            if (this.saldo >= cantidad) {
                this.saldo = this.saldo - cantidad;
            } else {
                throw new NoFondosDisponiblesException("No hay suficientes fondos");
            }
        }
    }

    public void depositar(double cantidad){
        if(cantidad > 0){
            this.saldo = this.saldo + cantidad;
        }
    }
}
```

CuentaBancaria.java

Segundo la subclase (en el fichero proporcionado se incluye mas código que se utilizará en ejemplos posteriores):

```
package fp2.poo.practica7;

public class CuentaAhorro extends CuentaBancaria {

    private double interes;

    public CuentaAhorro (){
        System.out.println("Constructor de CuentaAhorro");
        this.interes = 0.05d;
    }

    public void setInteres(double interes){
        this.interes = interes;
    }

    public double getInteres(){
        return this.interes;
    }
}
```

CuentaAhorro.java

Tercero la clase que contiene el método **main**:

```
package fp2.poo.practica7;

public class Practica7Ejercicio01 {
    public static void main (String args[]){
        CuentaAhorro cuentaAhorro = new CuentaAhorro();
        cuentaAhorro.setSaldo(1000.00d);
        System.out.println("Saldo : " + cuentaAhorro.getSaldo());
        System.out.println("Interes : " + cuentaAhorro.getInteres());
    }
}
```

Practica7Ejercicio01.java

Para probar este ejemplo ejecute:

make -f makeP7 prueba01

La subclase **CuentaAhorro** incluye todos los miembros de su superclase **CuentaBancaria**, por lo que **cuentaAhorro** además del atributo de la superclase **saldo** tiene el atributo **interes**.

Aunque **CuentaBancaria** es una superclase de **CuentaAhorro**, también es una clase autónoma y totalmente independiente. Aunque una superclase tenga una subclase, esto no significa que la superclase no pueda ser utilizada por sí sola. Además, una subclase puede a su vez ser una superclase de otra subclase.

La forma general de la declaración de una **clase** que hereda de otra es la siguiente:

```
class NombreDeLaSubclase extends NombreDeLaSuperclase {
    //cuerpo de la clase
}
```

Una variable de la superclase puede referenciar a un objeto de la subclase

A una variable referencia de la superclase se le puede asignar una referencia a cualquier subclase derivada de dicha superclase. Esta es una característica de la herencia muy útil y que se suele utilizar en bastantes situaciones. Veamos el siguiente ejemplo:

```

package fp2.poo.practica7;

public class Practica7Ejercicio02 {
    public static void main (String args[]){
        /*
         * Referencia a la superclase
         */
        CuentaBancaria cuentaBancaria = null;
        /*
         * Creacion de un objeto de la subclase
         */
        CuentaAhorro cuentaAhorro = new CuentaAhorro();
        cuentaAhorro.setSaldo(1000.00d);
        System.out.println("Saldo : " + cuentaAhorro.getSaldo());
        System.out.println("Interes : " + cuentaAhorro.getInteres());
        /*
         * Variable de la superclase que referencia a un objeto de la subclase
         */
        cuentaBancaria = new CuentaAhorro();
        System.out.println();
        System.out.println("Saldo : " + cuentaBancaria.getSaldo());
        /*
         * Variable de la superclase no puede acceder a los
         * miembros añadidos en la subclase
         */
        //System.out.println("Interes : " + cuentaBancaria.getInteres());
    }
}

```

Practica7Ejercicio02.java

Para probar este ejemplo ejecute:

make -f makeP7 prueba02

La variable **cuentaAhorro** es una referencia de tipo **CuentaAhorro** (subclase), y **cuentaBancaria** es una referencia de tipo **CuentaBancaria** (superclase).

Como **CuentaAhorro** es una subclase de **CuentaBancaria**, es posible asignar a **cuentaBancaria** una referencia del tipo **CuentaAhorro**.

El tipo de la variable referencia determina qué miembros son accesibles. En este caso no se puede acceder a **getInteres()**, mediante la referencia a la superclase (**cuentaBancaria**), ya que este método pertenece a la subclase, y no es visible desde una referencia a la superclase.

Quite el comentario de la última línea de código de la clase **Practica7Ejercicio02**, compile de nuevo y observe el resultado.

Uso de super en herencia

Cuando una subclase necesita referirse a su superclase inmediata, lo puede hacer utilizando la palabra clave **super**. La palabra reservada **super** se puede utilizar de dos formas:

1. La primera para llamar al constructor de la superclase.
2. La segunda se utiliza para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase.

A continuación vamos a ver el uso de **super** para llamar al constructor de la superclase. Una subclase puede llamar al constructor de la superclase utilizando **super** de la siguiente forma:

```
super (ListaDeParametros);
```

Aquí, **ListaDeParametros** especifica los parámetros del constructor de la superclase. Si se utiliza **super()**, tiene que ser la primera sentencia ejecutada dentro del constructor de la subclase.

Para ilustrar el ejemplo se ha añadido dos nuevos constructores a la clase **CuentaAhorro** en el primero se proporciona el saldo y en el segundo se proporciona el saldo y el interés.

```
package fp2.poo.practica7;

class CuentaAhorro extends CuentaBancaria {

    private double interes;

    public CuentaAhorro (){
        this.interes = 0.05d;
    }

    //Ejemplo de super para Practica7Ejercicio03
    public CuentaAhorro (double saldo){
        super(saldo);
        this.interes = 0.05d ;
    }

    public CuentaAhorro (double saldo, double interes){
        super(saldo);
        this.interes = this.interes;
    }

    public void setInteres(double interes){
        this.interes = interes;
    }

    public double getInteres(){
        return this.interes;
    }
}
```

CuentaAhorro.java

En este programa se utilizan los constructores añadidos:

```

package fp2.poo.practica7;

public class Practica7Ejercicio03 {
    public static void main (String args[]){
        /*
         * Llamada con un valor para saldo
         */
        CuentaAhorro cuentaAhorro1 = new CuentaAhorro(6000.d);
        System.out.println();
        System.out.println("Saldo de cuenta 1: " + cuentaAhorro1.getSaldo());
        System.out.println("Interes de cuenta 1: " + cuentaAhorro1.getInteres());
        System.out.println();
        /*
         * Llamada con un valor para saldo e interes
         */
        CuentaAhorro cuentaAhorro2 = new CuentaAhorro(5000.d, 0.05);
        System.out.println("Saldo de cuenta 2: " + cuentaAhorro2.getSaldo());
        System.out.println("Interes de cuenta 2: " + cuentaAhorro2.getInteres());
    }
}

```

Practica7Ejercicio03.java

Para probar este ejemplo ejecute:

make -f makeP7 prueba03

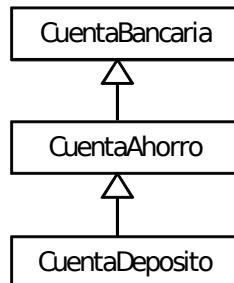
En este ejemplo se invoca al constructor de la superclase desde el constructor de la subclase mediante el uso de **super(saldo)**.

Para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase se utiliza **super.miembro**, siendo miembro, un atributo o un método visible a la superclase.

Creación de una jerarquía multinivel

Hasta ahora, sólo se han utilizado jerarquías de clases sencillas, formadas por una superclase y una subclase. Sin embargo, se pueden construir jerarquías que contengan tantos niveles de herencia como se desee.

Por ejemplo, dadas tres clases llamadas **CuentaBancaria**, **CuentaAhorro** y **CuentaDeposito**, **CuentaDeposito** es subclase de **CuentaAhorro**, **CuentaAhorro** es subclase de **CuentaBancaria**, tal y como se muestra a continuación.



Cuando se produce este tipo de situaciones, cada subclase hereda todas las características encontradas en todas sus superclases. En este caso, CuentaDeposito hereda todas las propiedades de CuentaAhorro y CuentaBancaria.

```
package fp2.poo.practica7;

/**
 * CuentaDeposito es una cuenta que por defecto si se deposita
 * mas de 1000 se incrementa el 5% de lo ingresado.
 * Tanto el umbral a partir del cual se aplica el interes, como
 * el interes son configurables en el constructor de la clase.
 */
public class CuentaDeposito extends CuentaAhorro {
    private double minimoAIngresar;
    private double interesIngreso;

    public CuentaDeposito (){
        super( 0.d, .02d );
        this.minimoAIngresar = 1000.0;
        this.interesIngreso = 5.0;
    }

    public CuentaDeposito (double minimoAIngresar ){
        super( 0.d, .02d );
        this.minimoAIngresar = minimoAIngresar ;
        this.interesIngreso = 5.0;
    }

    public CuentaDeposito (double minimoAIngresar , double interesIngreso ){
        super( 0.d, .02d );
        this.minimoAIngresar = minimoAIngresar ;
        this.interesIngreso = interesIngreso ;
    }

    public CuentaDeposito (double minimoAIngresar , double interesIngreso ){
        this.minimoAIngresar = minimoAIngresar ;
        this.interesIngreso = interesIngreso ;
    }

    public void depositar(double cantidad){
        if(cantidad >= this.minimoAIngresar){
            setSaldo( getSaldo() + cantidad + cantidad*this.interesIngreso/100);
        } else {
            super.depositar(cantidad);
        }
    }
}
```

CuentaDeposito.java

```
package fp2.poo.practica7;

public class Practica7Ejercicio04 {
    public static void main (String args[]){
        CuentaDeposito cuenta = new CuentaDeposito ();
        cuenta.depositar(10000.);
        System.out.println("Valor de saldo de la Cuenta Deposito "+cuenta.getSaldo() );
    }
}
```

Practica7Ejercicio04.java

Para probar este ejemplo ejecute:

make -f makeP7 prueba04

Gracias a la herencia, **CuentaDeposito** puede hacer uso de las clases **CuentaAhorro** y **CuentaBancaria**, definidas previamente, y añadir únicamente la información extra que necesita para su propio y específico uso. Ésta es una de las ventajas de la herencia: permite la reutilización de código.

Este ejemplo muestra otro aspecto importante: **super()** siempre hace referencia al constructor de la superclase más próxima. Dentro de **CuentaDeposito**, **super()** llama al constructor de **CuentaAhorro**. Dentro de **CuentaAhorro**, **super()** llama al constructor de **CuentaBancaria**.

Orden de ejecución de los constructores

Cuando se crea una jerarquía de clases, ¿en qué orden se ejecutan los constructores de cada una de las clases que constituyen la jerarquía?

La respuesta es que en una jerarquía de clases, **los constructores se ejecutan en orden de derivación**, desde la superclase a la subclase.

Además, como **super()** tiene que ser la primera sentencia que se ejecute dentro de constructor de la subclase, este orden es el mismo tanto si se utiliza **super()** como si no. Si no se utiliza **super()**, entonces se ejecuta el constructor por defecto o sin parámetros de cada superclase.

Sobreescritura de un método

En una jerarquía de clases, cuando un método de una subclase tiene el mismo nombre y tipo que un método de su superclase, entonces se dice que el método de la subclase **sobrescribe** al método de la superclase. Cuando se llama a un método sobrescrito dentro de una subclase, siempre se refiere a la versión del método definida por la subclase. La versión del método definida por la superclase está oculta. Si se desea acceder a la versión de la superclase de una función sobrescrita, se puede hacer utilizando **super**. La sobrescritura de métodos se produce **únicamente** cuando los nombres y tipos de los dos métodos son idénticos. Si no ocurre esto, el método está sobrecargado.

En el ejemplo anterior el método **depositar** de la clase **CuentaDeposito**, sobrescribe el método **depositar** de la clase **CuentaBancaria**, para llamar al método **depositar**, de **CuentaBancaria** se ha utilizado **super**.

Selección de método dinámica

La selección de método dinámica es el mecanismo mediante el cual una llamada a una función sobrescrita se **resuelve en tiempo de ejecución**, en lugar de durante la compilación. La selección de método dinámica es importante ya que es la forma que tiene Java de implementar el polimorfismo durante la ejecución.

Recordemos que **una variable de referencia de la superclase puede referirse a un objeto de la subclase**. Java utiliza esto para resolver durante la ejecución llamadas a métodos sobrescritos. Cuando un método sobrescrito se llama a través de una referencia de la superclase, Java determina la versión del método que debe ejecutar en función del tipo del objeto que está siendo referenciado en el momento en el que se produce la llamada. Esta decisión se realiza en tiempo de ejecución. Dependiendo del tipo de objeto referenciado se ejecutará una u otra versión de un método sobrescrito.

En otras palabras, es el **tipo del objeto que está siendo referenciado**, y no el tipo de la variable referencia, el que determina qué versión de un método sobrescrito será ejecutada. Así, si una superclase contiene un método que está sobrescrito en la subclase, entonces cuando son referenciados distintos tipos de objetos a través de una variable referencia de la superclase, se ejecutan distintas versiones del método.

A continuación se muestra un ejemplo que ilustra la selección de método dinámica.

```
package fp2.poo.practica7;

public class Practica7Ejercicio05 {
    public static void main (String args[]){
        /*
         * Referencia a la superclase
         */
        CuentaBancaria referenciaACuenta = null;
        /*
         * Objeto del tipo CuentaDeposito (subclase)
         */
        CuentaDeposito cuentaDeposito = new CuentaDeposito();
        /*
         * Objeto del tipo CuentaBancaria (superclase)
         */
        CuentaBancaria cuentaBancaria = new CuentaBancaria ();

        System.out.println("depositar en CuentaBancaria");
        referenciaACuenta= cuentaBancaria;
        referenciaACuenta.depositar(5000.);
        System.out.println(referenciaACuenta.getSaldo());

        System.out.println();
        System.out.println("depositar en CuentaDeposito ");
        referenciaACuenta= cuentaDeposito;
        referenciaACuenta.depositar(5000.);
        System.out.println(referenciaACuenta.getSaldo());
    }
}
```

Practica7Ejercicio05.java

Para probar este ejemplo ejecute:

```
make -f makeP7 prueba05
```

En este ejemplo se utiliza una referencia a la superclase (`CuentaBancaria`) para referirse a un objeto en primer lugar de la superclase (`CuentaBancaria`), y en segundo lugar para referirse a un objeto de la subclase (`CuentaDeposito`). En ambos la invocación al método es la misma (`referenciaACuenta.depositar(5000.)`). Sin embargo el resultado es diferente en cada llamada.

El tipo del objeto es el que determina qué método es invocado en la llamada (`referenciaACuenta.depositar(5000.)`). Cuando el tipo del objeto es `CuentaBancaria` se invoca al método `depositar` de `CuentaBancaria`. Cuando el tipo del objeto es `CuentaDeposito` se invoca al método `depositar` de `CuentaDeposito`.

Miembros static

Hay ocasiones en las que se necesita definir un miembro de una clase que será utilizado independientemente de cualquier objeto de esa clase. Normalmente, a un miembro de una clase se accede a través de la referencia a un objeto. Sin embargo, es posible crear un miembro que pueda ser utilizado por sí mismo sin referirse a una instancia específica. Para crear un miembro de ese tipo es necesario preceder su declaración con la palabra clave **static**.

Cuando se declara un miembro como **static**, se puede acceder a él antes de que se haya creado ningún objeto de esa clase, y sin hacer referencia a ningún objeto. Se pueden declarar **static** tanto los métodos como las variables. El ejemplo más común de un miembro **static** es `main()`. `main()` tiene que ser declarado como **static** ya que es llamado antes de que exista ningún objeto.

Las variables de instancia declaradas como **static** pueden ser accedidas mediante el nombre de la clase en vez de la referencia al objeto.

Cuando se declaran objetos de una clase, no se hace ninguna copia de las variables **static**. De hecho, todas las instancias de la clase comparten la misma variable **static**.

Clases abstractas

Una característica extremadamente útil de la programación orientada a objetos es el concepto de **clase abstracta**. Utilizando clases abstractas podemos declarar clases que definen sólo parte de una implementación, dejando a las clases que heredan de la clase abstracta la tarea de proporcionar las implementaciones específicas de algunos métodos, o de todos.

Una clase **no abstracta** tiene implementados todos los métodos de las interfaces que implemente, y los métodos abstractos de las clases abstractas de las que herede, implementados en dicha clase no abstracta o en alguna de sus superclases. Además puede añadir nuevos métodos no abstractos.

Las clases abstractas definen una superclase que declara la estructura de la clase dada sin proporcionar la implementación completa de todos los métodos. Y por tanto, definen una superclase de forma generalizada que será compartida por todas las subclases, dejando a cada subclase la tarea de completar los detalles de la implementación.

Las clases abstractas son útiles cuando parte del comportamiento está definido para la mayoría o todos los objetos de un tipo dado, pero algo del comportamiento sólo tiene sentido para la clase particular y no para una superclase general. Una clase así se declara como **abstract**, y los métodos no implementados en esa clase se marcan también como **abstract**.

```
abstract tipo nombre(ListaDeParametros);
```

Como se puede observar, este método no tiene cuerpo (implementación).

Cualquier clase que contenga uno o varios métodos abstractos también se tiene que declarar como **abstract**. Para declarar una clase abstracta, simplemente se utiliza la palabra clave **abstract** al principio de la declaración de la clase.

No se pueden crear instancias de clases abstractas con el operador **new**. No se pueden declarar constructores **abstract** o métodos **abstract** estáticos. Cualquier subclase de una clase abstracta debe implementar todos los métodos abstractos de la superclase o ser declarada también como **abstract**.

Las clases abstractas pueden añadir tantos atributos o métodos como sea necesario. Aunque no se pueden crear instancias de las clases abstractas, éstas pueden ser utilizadas para crear referencias a objetos, ya que la aproximación de Java al polimorfismo dinámico se implementa utilizando referencias de la superclase. Es posible crear una referencia de una clase abstracta para que pueda ser utilizada como referencia a un objeto de una subclase.

Uso de final con la herencia

La palabra **final** tiene tres usos.

1. En primer lugar, se puede utilizar para crear el equivalente de una constante con nombre. Habitualmente se suelen utilizar las interfaces para este propósito. Por ejemplo,

```
package fp2.poo.practica7;
```

```
public interface Practica7Ejercicio06 {  
    public final static int VALOR_MAXIMO = 100;  
}  
  
class Main{  
    public static void main (String args[]){  
        System.out.println("Valor de Practica7Ejercicio06.VALOR_MAXIMO :"  
                           + Practica7Ejercicio06.VALOR_MAXIMO );  
    }  
}
```

Practica7Ejercicio06.java

2. El uso de **final** con la herencia tiene dos usos:

- A. Evitar la sobrescritura de método: Aunque la sobrescritura de métodos es una de las características más poderosas de Java, hay ocasiones en las que es conveniente evitar que esto ocurra. Cuando se desea que un método no pueda ser sobrescrito, es necesario utilizar la palabra clave **final** como modificador al principio de su declaración. Los métodos como **final** no pueden ser sobrescritos.

Ejercicio: marque como **final** el método **depositar (public final void depositar(double cantidad))**, de la clase **CuentaBancaria** y ejecute de nuevo la prueba 4: Para probar este ejemplo ejecute:

```
make -f makeP7 prueba04
```

Obteniendo un error en compilación, ya que se intenta sobrescribir un método **final**.

- B. Uso de **final** para evitar la herencia. A veces se desea evitar que una clase pueda ser heredada. Para hacer esto, se utiliza la palabra clave **final** en la declaración de la clase. Cuando se declara una clase como **final**, implícitamente se están declarando todos sus métodos como **final**. Como se podría esperar, no es válido declarar una clase como **abstract** y **final**, ya que una clase abstracta es incompleta por sí misma y necesita que sus subclases proporcionen las implementaciones completas.

Ejercicio a entregar:

Se proporciona la siguiente interfaz.

Figura.java

```
/***
 *  @(#)Figura.java
 *
 *  Fundamentos de Programacion II. GIT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *  Marzo-2016
 *  Modificado Abril 2018
 */

package fp2.poo.practica7.utilidades;

/***
 *  Descripción: interfaz para la gestión de figuras geométricas
 *
 *  @author Fundamentos de Programacion II. GIT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *  @since Marzo-2016
 *  @version 1.0
 *
 */
public interface Figura {

    /**
     * Devuelve el área de la figura como un float
     * @return area, tipo float
     */
    float getArea();
    /**
     * Devuelve el perímetro de la figura como un float
     * @return perimetro, tipo float
     */
    float getPerimetro();
    /**
     * Devuelve el color de la figura como un String
     * @return area, tipo String
     */
    String getColor();
} //Cierre de la interfaz
```

Se proporciona la siguiente clase abstracta

MiFigura.java

```
/*
 *  @(#)MiFigura.java
 *
 *  Fundamentos de Programacion II. GIT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *
 */

/**
 * Descripcion: Implementación de la interfaz Figura genérica, para métodos
 * comunes, en este caso solamente el método toString, para mostrar las
 * clases
 * como interesa.
 *
 * @version 1.0 Marzo 2016 (Modificado Abril 2018)
 * @author Fundamentos de Programacion II
 */
package fp2.poo.practica7.utilidades;

public abstract class MiFigura implements Figura {

    /**
     * Sobrescritura del método toString, que facilita la depuración
     * y análisis del código.
     *
     * @return la figura como un String, tipo String
     * @see java.lang.Object#toString
     */
    public String toString(){
        int indicepunto=this.getClass().getName().lastIndexOf(".");
        return this.getClass().getName().substring(indicepunto+1)
            + " de color " + this.getColor() + "\n";
    }

    /**
     * Devuelve el área de la figura como un float
     * @return area, tipo float
     */
    abstract float getArea();

    /**
     * Devuelve el perímetro de la figura como un float
     * @return perimetro, tipo float
     */
    abstract public float getPerimetro();

    /**
     * Devuelve el color de la figura como un String
     * @return area, tipo String
     */
    abstract public String getColor();
}
```

Complete la implementación de la clase **Círculo** que se muestra parcialmente a continuación.

Círculo.java

```
package fp2.poo.practica7.utilidades;
import java.awt.Color;
import java.util.*;

public class Círculo extends MiFigura{

    private float diametro;
    private float PI = 3.1416f;
    private Color color;

    // Resto del código
}
```

Complete la implementación de la clase **Cuadrado** que se muestra parcialmente a continuación.

Cuadrado.java

```
package fp2.poo.practica7.utilidades;
import java.awt.Color;

public class Cuadrado extends MiFigura {

    private float lado;
    private Color color;

    // Resto del código
}
```

Implemente la clase **Main** en el paquete **fp2.poo.practica7.XXX** (siendo XXX el uvus) para crear objetos del tipo **Círculo** y **Cuadrado**.

Comprima el código en un archivo de nombre el uvus del alumno y entréguelo en la Actividad de la práctica.



El paquete java.io

OBJETIVO

En esta práctica se presenta el paquete `java.io` para realizar las operaciones de entrada y salida de datos, incluyendo las operaciones de lectura/escritura en archivos.

1 Introducción

El lenguaje Java proporciona una serie de clases para facilitar las tareas de lectura y escritura de ficheros. Estas clases consideran muchos de los casos que se pueden dar en este tipo de operaciones.

Los programas Java realizan la entrada y salida de datos a través de **flujos** (streams). Un flujo es una abstracción que produce o consume información. Los flujos están relacionados con los dispositivos físicos a través del sistema de entrada/salida de Java. Todos los flujos tienen un comportamiento similar, incluso aunque estén relacionados con distintos dispositivos físicos. Por esta razón, se pueden aplicar las mismas clases y métodos de entrada/salida a cualquier tipo de dispositivo. Esto significa que un flujo de entrada puede abstraer distintos tipos de entrada, desde un archivo de disco a un teclado o a otro dispositivo. Del mismo modo, un destino de salida puede ser una consola, o un archivo de disco. Los flujos son una manera limpia de tratar con la entrada/salida sin necesitar que el código comprenda la diferencia entre los distintos dispositivos.

En el lenguaje de programación C, los flujos (o stream) también se usan con las operaciones de entrada y salida, tanto para el manejo de ficheros como para realizar la entrada y salida, mediante los 3 flujos predefinidos de tipo `FILE *` (`stdin`, `stdout` y `stderr`). La aproximación de Java a los flujos es bastante similar a la de C.

2 Clasificación inicial.

A la hora de trabajar con los ficheros de datos, y como hacen otros lenguajes, Java considera dos tipos:

- *Ficheros binarios*: contienen cualquier tipo de dato en general. Estos tipos de ficheros pueden contener cualquier tipo de dato: números en cualquier codificación, caracteres de texto, objetos, o/y combinaciones de los anteriores, y usan unidades de datos de 8 bits.
- Ficheros cuyo contenido se identifica con elementos imprimibles: números, letras (mayúsculas o minúsculas), signos de puntuación,... Son los que comúnmente se conocen como *ficheros de texto*, son datos que se pueden contener en los tipos de datos `char` o `String`, y que usan unidades de datos de 16 bits.

Java proporciona una gran cantidad de clases para la lectura, creación y escritura de ficheros, que se concentran en el paquete `java.io`. Los casos en los que se pueden utilizar son muy amplios.

Para poder utilizar este código, los ficheros Java deben incluir al principio la línea:

```
import java.io.*;
```

3 Flujos predefinidos

Como ya sabemos, todos los programas Java importan automáticamente el paquete `java.lang`. Este paquete define una clase llamada `System` que encapsula algunos aspectos del entorno de ejecución. Por ejemplo, utilizando algunos de sus métodos se puede obtener la hora actual o los valores de algunas propiedades asociadas con el sistema. Además, contiene tres variables con flujos predefinidos llamadas `in`, `out` y `err`. Estas variables están declaradas como `public` y `static` en `System`. Esto significa que se pueden utilizar en cualquier parte del programa y sin tener una referencia a un objeto `System` específico.

`System.out` se refiere al flujo de salida estándar. Por defecto, es la consola. `System.in` hace referencia a la entrada estándar, que es, por defecto, el teclado. `System.err` se refiere al flujo de error estándar, que, por defecto, también es la consola. Sin embargo, estos flujos pueden ser redirigidos a cualquier otro dispositivo de E/S compatible.

En las prácticas anteriores ya se ha utilizado `System.out`. De igual forma se puede utilizar `System.err`. En esta práctica veremos el uso de `System.in`.

4 Ficheros binarios.

Java implementa los flujos dentro de una jerarquía de clases definida en el paquete `java.io`. En la parte superior de la jerarquía hay dos clases abstractas: `InputStream` y `OutputStream`. Java tiene algunas subclases concretas de cada una de ellas para gestionar las diferencias que existen entre los distintos dispositivos, como archivos de disco, conexiones de red y búferes de memoria.

Las clases abstractas `InputStream` y `OutputStream` definen algunos métodos que las otras clases implementan. Dos de los métodos más importantes son `read()` y `write()`, que respectivamente lee bytes del flujo y escribe bytes en el flujo. Ambos métodos están declarados como abstractos dentro de `InputStream` y `OutputStream` y son implementados en las clases derivadas.

Vamos a presentar la forma de trabajar con ficheros binarios. Todas las operaciones con este tipo de ficheros giran en torno a dos clases: `FileInputStream` para lectura de datos de un fichero binario, y `FileOutputStream` para la escritura. También vamos a ver una serie de clases que complementan el funcionamiento de las dos mencionadas añadiendo funcionalidades interesantes para mejorar el rendimiento de los programas y la facilidad de uso de la interfaz por parte del usuario.

4.1 Lectura de ficheros binarios.

Existe una clase básica que implementa las funciones de lectura, denominada `java.io.FileInputStream` con unos métodos que permiten leer a nivel de byte, y luego una serie de clases que complementan su funcionamiento añadiendo mejoras de rendimiento o de flexibilidad a la hora de realizar las lecturas.

- La clase `FileInputStream` tiene varios constructores, siendo el más interesante el constructor al que se le pasa un `String` con la ruta al fichero que se quiere leer:

```
FileInputStream fich = new FileInputStream("./lista.bin");
```

- Los métodos principales son los siguientes:
 - `close()`: cierra el fichero liberando todos los recursos asociados a la lectura.
 - `read`: método para la lectura a nivel de byte. Tiene las siguientes variantes:
 - `read()`: lee un byte del fichero y lo devuelve como resultado de la operación. Si no hay caracteres devuelve -1 si es el final del fichero.
Ejemplo:

```
byte datoLeido = fich.read();
```
 - `read(byte[] buffer, int despl, int long)`: lee como máximo `long` bytes y los almacena en `buffer` a partir de la posición `despl`. Devuelve el número de bytes leídos o -1 si no ha podido leer nada por fin de fichero.
 - `read(byte[] buffer)`: lee bytes y los guarda en `buffer`. Como máximo lee la longitud de `buffer`. Devuelve el número de bytes leídos o -1 si no ha podido leer nada por fin de fichero.

Estos no son los únicos métodos de esta clase, algunas clases más se presentan en la siguiente tabla.

Clase	Significado
BufferedInputStream	Flujo de entrada con búfer
BufferedOutputStream	Flujo de salida con búfer
ByteArrayInputStream	Flujo de entrada que lee una matriz de bytes
ByteArrayOutputStream	Flujo de salida que escribe una matriz de bytes
DataInputStream	Flujo de entrada que contiene métodos para leer los tipos básicos de Java.
DataOutputStream	Flujo de salida que contiene métodos para escribir los tipos básicos de Java.
FileInputStream	Flujo de entrada que lee de un archivo
FileOutputStream	Flujo de salida que escribe en un archivo
FilterInputStream	Implementa a InputStream para realizar filtrado.
FilterOutputStream	Implementa a OutputStream para realizar filtrado.
InputStream	Clase abstracta que define un flujo de entrada
LineNumberInputStream	Flujo de entrada que contiene líneas.
OutputStream	Clase abstracta que define un flujo de salida
PipedInputStream	Flujo de entrada de tipo Piped.
PipedOutputStream	Flujo de salida del tipo Piped.
PrintStream	Flujo de salida que contiene los métodos <code>print()</code> y <code>println()</code> .
PushbackInputStream	Flujo de entrada que, una vez leído un byte, permite que se devuelva de nuevo al flujo de salida.
RandomAccessFile	Permite acceso aleatorio a un archivo de E/S.
SequenceInputStream	Flujo de entrada que es una combinación de dos o más flujos de entrada que serán leídos secuencialmente, uno después de otro.
StreamTokenizer	Divide el flujo de entrada en símbolos delimitados por conjuntos de caracteres.
StringBufferInputStream	Flujo de entrada que lee de una cadena.

Algunas de las clases de este paquete para el manejo de bytes que son de utilidad son las siguientes:

- o **BufferedInputStream**: esta clase facilita el problema de velocidad de ejecución por acceso a disco físico implementando una memoria caché intermedia de donde leer los datos. Al constructor se le pasa un objeto de tipo `InputStream`, o cualquier otro objeto que herede de él, como es `FileInputStream`. No hay métodos adicionales de interés a la hora de leer tipos de datos.
- o **DataInputStream**: añade métodos que permiten leer tipos de datos primitivos de java, como pueden ser `int`, `long`, `short`, `float`, `double`, ... Tiene un solo constructor al que se le pasa un objeto de tipo `InputStream`, o cualquier otro objeto que herede de él, como es `FileInputStream` o `BufferedInputStream`. En cuanto a los métodos añade uno por cada tipo de dato primitivo que lee:
 - `readBoolean()`: devuelve un valor de tipo booleano leído del fichero.
Ejemplo:

`boolean bandera = fich.readBoolean();`

▪ `readByte()`: devuelve un valor de tipo byte leído del fichero.

- `readDouble()`: devuelve un valor de tipo `double` leído del fichero.
- `readFloat()`: devuelve un valor de tipo `float` leído del fichero.
- `readInt()`: devuelve un valor de tipo `int` leído del fichero.
- `readShort()`: devuelve un valor de tipo `short` leído del fichero.
-

La forma de crear un objeto que nos permitiera leer de un fichero utilizando un buffer intermedio para mejorar la velocidad de acceso y que nos permitiera leer tipos primitivos de java sería la siguiente:

```
FileInputStream fich = new FileInputStream("./lista.bin");
BufferedInputStream bis = new BufferedInputStream(fich);
DataInputStream dis = new DataInputStream(bis);

...
int datoEntero = dis.readInt();
```

4.2 Escritura de ficheros binarios

La clase principal que nos permite crear y/o añadir contenidos a un fichero binario es `java.io.FileOutputStream`. Esta clase nos permite escribir en un fichero a nivel de byte.

La clase cuenta con varios constructores siendo de interés para esta práctica los siguientes:

- `FileOutputStream(String nombreFichero)`: se le pasa como argumento un `String` con el camino al fichero a crear o abrir para añadir contenido. Si el fichero no existe lo crea, y si existe lo borra y lo crea de nuevo. Ejemplo:

```
FileOutputStream fich =new FileOutputStream("./listado.txt");
```

- `FileOutputStream(String nombreFichero, boolean append)`: se le pasa como argumento un `String` con el camino al fichero a crear o abrir para añadir contenido y una variable de tipo `boolean` que indica si se añade el contenido al final del fichero en caso de que ya exista el mismo o no. Ejemplo:

```
FileOutputStream fich =new FileOutputStream("./listado.txt", true);
```

Los métodos principales de esta clase son:

- `close()`: cierra el fichero liberando todos los recursos asociados a la escritura.

- `write`: método para la escritura de caracteres. Las posibilidades de uso son las siguientes:

- `void write(int c)`: escribe el byte representado por `c`. Ejemplo:

```
int datoAEscribir = 0xA3;  
fich.write(datoAEscribir);
```

- `void write(byte[] buffer, int desp, int long)`: escribe parte de los bytes incluido en la tabla de `byte` referenciada por `buffer`, comenzando desde la posición `desp`, hasta un total de `long` bytes.

- `void write(byte[] buffer)`: escribe todos los bytes incluidos en `buffer`.

También existen una serie de clases complementarias que aumentan las funcionalidades de esta clase. Las más interesantes para esta práctica son las siguientes:

- `BufferedOutputStream`: esta clase aumenta la velocidad de ejecución de las aplicaciones usando una memoria caché intermedia y reduciendo el número de accesos físicos para escritura al disco. Al constructor se le pasa un objeto de tipo `OutputStream`, o cualquier otro objeto que herede de él, como es `FileOutputStream`. No hay métodos adicionales de interés a la hora de escribir tipos de datos.

- o **DataOutputStream**: incrementa la interfaz de la clase de escritura a ficheros añadiendo métodos para escribir tipos de datos primitivos de java, como pueden ser **int**, **long**, **short**, **float**, **double**, . . . Tiene un solo constructor al que se le pasa un objeto de tipo **OutputStream**, o cualquier otro objeto que herede de él, como es **FileOutputStream** o **BufferedOutputStream**. En cuanto a los métodos añade uno por cada tipo de dato primitivo:

- **writeBoolean(boolean dato)**: escribe el valor **dato** de tipo **booleano** en el fichero. Ejemplo:

```
boolean bandera = false;
fich.writeBoolean(bandera);
```

- **writeByte(byte dato)**: escribe el valor de dato de tipo **byte** en el fichero.
- **writeDouble(double dato)**: escribe el valor de dato de tipo **double** en el fichero.
- **writeFloat(float dato)**: escribe el valor de dato de tipo **float** en el fichero.
- **writeInt(int dato)**: escribe el valor de **dato** de tipo **int** en el fichero.
- **writeShort(short dato)**: escribe el valor de dato de tipo **short** en el fichero.
-

La forma de crear un objeto en java que aprovechara las ventajas de estas dos clases sería la siguiente:

```
FileOutputStream      fich = new FileOutputStream("./lista.bin");
BufferedOutputStream bos  = new BufferedOutputStream(fich);
DataOutputStream      dos  = new DataOutputStream(bos);

...
int datoEntero = 12500;
dos.writeInt(datoEntero);
```

5 Ficheros de texto.

Tal y como ya se ha indicado Java implementa los flujos dentro de una jerarquía de clases definida en el paquete `java.io`. En la parte superior de la jerarquía hay dos clases abstractas: `InputStream` y `OutputStream`, para el manejo de flujos de 8 bits. Y a partir de estas clases Java tiene algunas subclases concretas de cada una de ellas para gestionar las diferencias que existen entre los distintos dispositivos, como archivos de disco, conexiones de red y búferes de memoria.

Para el manejo de flujos de 16 bits o flujo de texto la organización es similar. En este caso las clases abstractas son, `Reader` y `Writer`, que contienen métodos similares a las clases abstractas `InputStream` y `OutputStream`, pero en este caso las unidades mínimas gestionadas son de 16 bits.

5.1 Ficheros de Texto

Si para la lectura de ficheros binarios todas las operaciones con este tipo de ficheros giran en torno a dos clases: `FileInputStream` para lectura de datos de un fichero binario, y `FileOutputStream` para la escritura. Para los ficheros de texto las operaciones giran en torno a `FileReader` y `FileWriter`. Por tanto, para la lectura, creación y escritura de ficheros se utilizan las siguientes clases del paquete `java.io`: `java.io.FileReader` y `java.io.FileWriter` y un conjunto de clases asociadas con estas.

5.2 Lectura de ficheros de texto

La clase `java.io.FileReader` (o simplemente `FileReader` si hemos incluido la sentencia `import` indicada en un apartado anterior) proporciona una serie de métodos para una lectura básica de un fichero de texto.

Esta clase tiene tres constructores, de los cuales vamos a utilizar el constructor al que se le pasa una cadena de texto en un `String` con la ruta al fichero que queremos abrir. Un ejemplo de cómo se instancia un objeto de esta clase es el siguiente:

```
FileReader fich = new FileReader("./listin.txt");
```

Al ejecutar esta sentencia y si todo ha ido bien tenemos una referencia al fichero que podemos utilizar para leer datos en forma de caracteres. Los métodos de estas clases que son de interés son los siguientes:

- `close()`: cierra el fichero liberando todos los recursos asociados a la lectura.
- `read`: método para la lectura de caracteres. Tiene las siguientes variantes:
 - `read()`: lee un carácter del fichero y lo devuelve como resultado de la operación. Si no hay caracteres devuelve -1 si es el final del fichero. Ejemplo:

```
System.out.println("caracter leido:"+fich.read());
```
 - `read(char[] buffer, int despl, int long)`: lee como máximo `long` caracteres y los almacena en `buffer` a partir de la posición `despl`. Devuelve el número de caracteres leídos o -1 si no ha podido leer nada por fin de fichero.

- o `read(char[] buffer)`: lee caracteres y los guarda en `buffer`. Devuelve el número de caracteres leídos o -1 si no ha podido leer nada por fin del fichero.

Estos no son los únicos métodos de esta clase, si quiere conocer el resto consulta la documentación del paquete java disponible en Internet.

La clase `FileReader` presenta dos problemas:

- Solo permite leer caracteres como unidad de referencia, no permite otros métodos de lectura. Como hemos visto solo presenta un método `read` para acceder a los datos.
- Esta clase cada vez que necesita datos accede físicamente al disco. No utiliza ningún buffer intermedio de datos, ralentizando mucho el acceso al fichero y por tanto el rendimiento de las aplicaciones.

Para solucionar estos dos problemas se utiliza otra clase que complementa las funcionalidades de `FileReader`: `java.io.BufferedReader`. Esta clase utiliza un Buffer o memoria intermedia antes de acceder a disco por lo que mejora el rendimiento de los accesos. Esta clase tiene un constructor al que se le pasa como argumento el objeto de la clase `FileReader` al que complementa. Ejemplo:

```
BufferedReader fichBuff = new BufferedReader(fich);
```

En lo que se refiere a métodos de la clase, incluye la posibilidad de leer líneas de caracteres completas utilizando el método `readLine()`, al que no se le pasan argumentos y devuelve un `String` con la línea leída del fichero. Ejemplo:

```
String lineaLeida = fichBuff.readLine();
```

5.3 Escritura de Ficheros de Texto.

La clase que se utiliza para la escritura de fichero de texto es `java.io.FileWriter`. El funcionamiento es idéntico al de `FileReader` pero orientado a la escritura de ficheros:

- Respecto al constructor, la clase tiene varios constructores posibles, pero nosotros vamos a considerar dos de ellos a la hora de hacer esta práctica:
 - o `FileWriter(String nombreFichero)`: se le pasa como argumento un `String` con el camino al fichero a crear o abrir para añadir contenido. Si el fichero no existe lo crea, y si existe lo borra y lo crea de nuevo. Ejemplo:


```
FileWriter fich = new FileWriter("./listado.txt");
```
 - o `FileWriter(String nombreFichero, boolean append)`: se le pasa como argumento un `String` con el camino al fichero a crear o abrir para añadir contenido y una variable de tipo `boolean` que indica si se añade el contenido al final del fichero en caso de que ya exista el mismo o no. Ejemplo:


```
FileWriter fich = new FileWriter("./listado.txt", true);
```

- Los métodos principales de esta clase son:
- `close()`: cierra el fichero liberando todos los recursos asociados a la escritura.
- `write`: método para la escritura de caracteres. Tenemos las siguientes posibilidades:

- o `void write(int c)`: escribe el carácter representado por `c`.
- o `void write(char[] buffer, int desp, int long)`: escribe parte de los caracteres incluidos en la tabla de `char` referenciada por `buffer`, comenzando desde la posición `desp`, hasta un total de `long` caracteres.
- o `void write(String texto, int desp, int long)`: escribe parte de los caracteres incluidos en el `String` referenciado por `texto`, comenzando desde la posición `desp`, hasta un total de `long` caracteres.
- `FileWriter` tiene los mismos dos problemas que vimos con `FileReader` en lo que se refiere a acceso continuo a disco con los problemas de ralentización de los programas, así como que sus métodos no manejan el concepto de línea.

Sí presenta una diferencia en cómo propone soluciones para estas cuestiones, mientras que en el caso anterior usando una única clase `BufferedReader` de complemento solucionábamos las dos circunstancias, en el caso de escritura es necesario utilizar dos clases complementarias, resolviendo cada una de ellas uno de los problemas:

- o `BufferedWriter`: esta clase nos soluciona el problema de los accesos continuos al fichero, al utilizar una zona intermedia de memoria donde acumula datos antes de grabarlos en disco. Al constructor se le pasa un objeto de tipo `FileWriter`. Los métodos son los mismos de `FileWriter`.
- o `PrintWriter`: esta clase proporciona métodos adicionales al `write` para escribir datos en un fichero, incluidos los que introducen el carácter separador de línea. Al constructor se le pasa un objeto de tipo `Writer` (como por ejemplo `BufferedWriter` o `FileWriter`). En cuanto a métodos de escritura en ficheros podemos destacar los siguientes:
- `println(char[] buffer)`: escribe en el fichero los caracteres incluidos en la tabla de caracteres `buffer`, y añade un salto de línea.
- `println(String cadena)`: escribe el `String` `cadena` en el fichero y un salto de línea.

La utilización de las tres clases de manera combinada se haría de la siguiente manera:

```
FileWriter fich = new FileWriter("./listado.txt", true);
BufferedWriter bw = new BufferedWriter(fich);
PrintWriter pw = new PrintWriter(bw);
```

6 Tratamiento de errores

A la hora de trabajar con acceso a disco y ficheros se pueden producir muchas circunstancias que pueden dar lugar a errores: intentar leer de ficheros que no existen, intentar crear ficheros que ya están creados, intentar leer un fichero al que no se tiene permiso de acceso, intentar escribir en un fichero pero no hay espacio libre en disco,....

Para tratar todos los posibles errores y situaciones anómalas que puedan producirse en la gestión de ficheros se utiliza el mecanismo de excepciones estándar de java. Las excepciones que se generan en caso de errores son del tipo `IOException` o de alguna clase que hereda de esta (como `FileNotFoundException` ó `EOFException`).

Para poder tratar estas excepciones debemos utilizar las estructuras de control de java `try {...}` `catch {...}` `finally {...}` explicado en una práctica anterior.

Siempre que se trabaja con clases que manejan ficheros, es muy importante para mantener la consistencia de estos ficheros (especialmente cuando trabajamos con zonas de memoria intermedia para mejorar el rendimiento) en cerrar los flujos que generan dichas clases una vez finalizado el uso de los mismos. Este cierre se debe garantizar en cualquier circunstancia, tanto si el funcionamiento ha sido correcto como si se ha producido alguna excepción que ha provocado el final del flujo normal deseable de nuestra aplicación. Es por eso que estas operaciones de cierre se ejecutan siempre dentro de la cláusula `finally`.

No obstante la ejecución del método `close()` que realiza este cierre de los flujos, también puede generar una excepción del tipo `IOException` si encuentra algún problema cuando va a cerrar los flujos. Para poder capturar y tratar esta excepción es necesario utilizar nuevamente una estructura `try {...}` `catch {...}` anidada con la que estamos tratando. De esta forma la estructura para la gestión de esta situación sería la siguiente:

```
try {  
    ...  
    FileWriter fichero = new FileWriter("~/home/alumno/ejemplo.txt");  
    ...  
} catch (IOException error1) {  
    ...(Tratamiento error)...  
}finally {  
    try {  
        fichero.close();  
    } catch (Exception error2) {  
        ...(Tratamiento error de cierre de fichero)...  
    }  
}
```

Ejemplos entregados.

1. Utilice la herramienta **make** para compilar los ejemplos que se van a ver a continuación de la siguiente forma.

make compila

y ejecute el código de este primer ejemplo en Java de la siguiente forma:

make ejecuta01

Nota: a partir de ahora en la mayor parte de las ejecuciones, cada prueba de los ejemplos se realiza mediante **make ejecutaXX**, con **XX** que indica el número del ejercicio que se está probando.

```
package fp2.poo.practica8;

import java.io.InputStream;
import java.io.IOException;

public class Practica8Ejercicio01 {
    public static void main(String args[]){
        InputStream in      = null;
        int cr              = 0;
        int total           = 0;
        int espacio         = 0;

        try{
            in = System.in;

            System.out.println("Introducción de datos desde la entrada estandar.");
            System.out.println("Introduzca datos y pulse la tecla ENTER.");
            System.out.println("La lectura de datos finaliza con el carácter 'q'.");

            for(total = 0; (cr = in.read()) != 'q' ; total++){
                if(Character.isWhitespace((char)cr))
                    espacio++;
            }

            in.close();
            System.out.println(total + " caracteres, " + espacio + " espacios");
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

En este ejemplo, se realiza la lectura de datos con flujos del tipo binario desde la entrada estándar. La lectura finaliza cuando se encuentra con el carácter ‘q’. Por pantalla se muestra el número total de caracteres leídos y el número de espacios en blanco introducidos desde la entrada estándar. Para comprobar si la información proporcionada desde el teclado es un espacio en blanco se utiliza el método estático **isWhitespace** de la clase **Character**.

2. Ejecute el siguiente ejemplo de la siguiente forma:

make ejecuta02

```
/**  
 * Descripcion: En este ejemplo se realiza la lectura de bytes de la entrada  
 * estandar. Es el mismo caso que el de Practica8Ejercicio01.java, pero en este  
 * caso el flujo de entrada se "envuelve" en un flujo de tipo InputStreamReader  
 * para realizar la conversion de 8 a 16 bits.  
 *  
 * version 1.0 Mayo 2014  
 * Fundamentos de Programacion II  
 */  
  
package fp2.poo.practica8;  
  
import java.io.InputStream;  
import java.io.IOException;  
import java.io.InputStreamReader;  
  
public class Practica8Ejercicio02 {  
    public static void main(String args[]){  
        InputStreamReader in = null;  
        int cr = 0;  
        int total = 0;  
        int espacio = 0;  
  
        try{  
            in = new InputStreamReader(System.in);  
            System.out.println("Introduccion de datos desde la entrada estandar.");  
            System.out.println("Introduzca datos y pulse la tecla ENTER.");  
            System.out.println("La lectura de datos finaliza con el caracter 'q'.");  
  
            for(total = 0; (cr = in.read()) != 'q' ; total++){  
                if(Character.isWhitespace((char)cr))  
                    espacio++;  
            }  
  
            in.close();  
            System.out.println(total + " caracteres, " + espacio + " espacios");  
        }catch(IOException e){  
            System.out.println(e);  
        }  
    }  
}
```

El resultado de la ejecución de este segundo ejemplo es el mismo que el del primero. En este caso se ha añadido la transformación de un flujo de 8 bit en uno de 16 bits, mediante el uso del constructor de la clase `InputStreamReader`. Esta clase construye una “envolvente” al flujo que se le proporciona como parámetro, en este caso `System.in`.

- 3.** Este ejemplo se puede ejecutar de dos formas: proporcionando un argumento en línea de comando que será el fichero a leer, o bien sin argumento en línea de comandos donde los datos se proporcionarán desde teclado. Para realizar la ejecución del programa sin argumentos en línea de comandos, teclee:

make ejecuta03SinFichero

Para realizar la ejecución del programa mediante un argumento en línea de comandos, teclee:

make ejecuta03ConFichero

Que invoca a:

java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio03

y
java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio03 ./ficherosEntrada/listado.txt

respectivamente.

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.OutputStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

/**
 * Recuenta el numero de bytes de un fichero.
 * Se proporciona el nombre como argumento en
 * linea de comandos. Sino se proporciona fichero se lee
 * de la entrada est ndar (System.in)
 */
public class Practica8Ejercicio03 {
    public static void main( String args[] ) {

        InputStream in = null;
        int total      = 0;

        try{
            if(args.length == 0){
                in = System.in;
            }else{
                in = new FileInputStream(args[0]);
            }
            int c = 0;

            while( (c = in.read()) != -1 ) {
                System.out.write((char)c);
                total++;
            }

            System.out.println("Numero total de caracteres leidos: " + total);
        }catch(FileNotFoundException e){
            System.out.println("fichero no encontrado "+e);
        }catch(IOException e){
            System.out.println(e);
        }finally {
            in.close();
        }
    }
}
```

En el caso de realizar la lectura del fichero, este se toma de **./ficherosEntrada/listado.txt**.

Nota: Para finalizar la lectura desde teclado pulse las teclas **Control+d**.

- 4.** Ejecute el siguiente ejemplo de la siguiente forma:

make ejecuta04

que ejecuta la siguiente orden:

```
java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio04 ./ficherosSalida/FicheroSalidaPrueba04.txt
```

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.OutputStream;
import java.io.FileOutputStream;

public class Practica8Ejercicio04 {

    public static void main(String[] args) {
        byte[] b = {'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', '\n'};
        OutputStream os = null;
        try {
            os = new FileOutputStream(args[0]);
            for (int i = 0; i < b.length; i++) {
                os.write(b[i]);
                System.out.print(" " + (char)b[i]);
            }
            os.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

En este ejemplo se realiza la escritura en un archivo utilizando clases que manejan información de tipo binario.

El nombre del archivo se proporciona como argumento en línea de comandos. El flujo es del tipo: `FileOutputStream`, y usa el método `write` para realizar la escritura.

5. Para realizar la ejecución del programa teclee:

make ejecuta05

La llamada se realiza de la siguiente forma:

```
java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio05 ./ficherosSalida/FicheroSalidaPrueba05.txt
```

```
package fp2.poo.practica8;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.io.FileOutputStream;

public class Practica8Ejercicio05 {
    public static void main(String args[]){
        InputStream in = null;
        OutputStream out = null;
        int cr = 0;

        try{
            in = System.in;
            out = new FileOutputStream(args[0]);

            for( ; (cr = in.read()) != -1 ; ){
                out.write((byte)cr);
                System.out.print("'" + (char)cr);
            }
            out.close();
            in.close();
        } catch ( IOException e ) {
            e.printStackTrace();
        }
    }
}
```

En este ejemplo se proporciona el nombre del fichero donde se va a realizar la escritura como argumento en línea de comandos (`./ficherosSalida/FicheroSalidaPrueba05.txt`).

Los datos leídos se proporcionan desde la entrada estándar. Y se muestran (a modo de eco) por pantalla.

Para finalizar pulse **Control-d**.

6 . Para realizar la ejecución del programa teclee:

make ejecuta06

La llamada se realiza de la siguiente forma:

```
java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio06      ./ficherosEntrada/listado.txt  
./ficherosSalida/FicheroSalidaPrueba06.txt
```

```
package fp2.poo.practica8;  
  
import java.io.InputStream;  
import java.io.OutputStream;  
import java.io.IOException;  
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.FileNotFoundException;  
import java.io.Reader;  
  
public class Practica8Ejercicio06 {  
    public static void main(String[] args) {  
  
        try {  
            InputStream in = new FileInputStream( args[0] );  
            OutputStream os = new FileOutputStream(args[1]);  
            int cr = 0;  
  
            System.out.println("Se han abierto los dos ficheros " + args[0] + " y " + args[1] );  
            System.out.println("Comienza la copia de un fichero en otro. ");  
  
            while((cr = in.read()) != -1 ) {  
                System.out.print("") + (char) cr);  
                os.write((byte)cr);  
            }  
  
        } catch ( FileNotFoundException e) {  
            System.err.println("Fichero no encontrado");  
        } catch (Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

Este ejemplo realiza la copia de un fichero de entrada y que existe en otro fichero de salida.

7. Para realizar la ejecución del programa teclee:

make ejecuta07

```
/**  
 * Descripción: Uso de un flujo buferado de un fichero.  
 *  
 *  
 * version 1.0 Mayo 2014  
 * Fundamentos de Programacion II  
 */  
  
package fp2.poo.practica8;  
  
import java.io.InputStream;  
import java.io.OutputStream;  
import java.io.IOException;  
import java.io.FileInputStream;  
import java.io.BufferedInputStream;  
import java.io.FileNotFoundException;  
import java.io.Reader;  
  
public class Practica8Ejercicio07 {  
    public static void main(String[] args) {  
        FileInputStream fin = null;  
        BufferedInputStream bis = null;  
        int c = 0;  
  
        try {  
            if (args.length == 1) {  
  
                fin = new FileInputStream(args[0]);  
                bis = new BufferedInputStream(fin);  
  
                while ((c = bis.read()) != -1) {  
                    System.out.print((char)c);  
                }  
            } else {  
                System.err.println("Proporcione el nombre del fichero.");  
            }  
        } catch (FileNotFoundException e) {  
            System.err.println(e);  
        } catch (IOException e) {  
            System.err.println(e);  
        } catch (Exception e) {  
            System.err.println(e);  
        }  
    }  
}
```

Este programa muestra un ejemplo de buferado de los datos de un fichero. El nombre del fichero se le proporciona como argumento en línea de comandos. El buferado se realiza proporcionándole como parámetro a la clase `BufferedInputStream`, el flujo de tipo `FileInputStream` asociado al archivo.

8. Para realizar la ejecución del programa teclee:

make ejecuta08

```
package fp2.poo.practica8;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;

public class Practica8Ejercicio08 {
    public static void main(String[] args) {

        FileInputStream      is      = null;
        BufferedInputStream   bis     = null;
        BufferedOutputStream  bos    = null;
        FileOutputStream      os     = null;

        int                  value = -1;

        try{
            is    = new FileInputStream( args[0] );
            bis   = new BufferedInputStream(is);

            os = new FileOutputStream ( args[1] );
            bos = new BufferedOutputStream(os);

            while ((value = bis.read()) != -1) {
                bos.write(value);
            }

            /* Invoca a flush para forzar que los bytes se escriban en el flujo. */
            bos.flush();
        }catch(IOException e){
            // if any IOException occurs
            e.printStackTrace();
        }finally{
            try{
                if(is!=null)
                    is.close();
                if(bis!=null)
                    bis.close();
                if(os!=null)
                    os.close();
                if(bos!=null)
                    bos.close();
            }catch(IOException e){
                // if any IOException occurs
                e.printStackTrace();
            }
        }
    }
}
```

En este ejemplo se realiza el buferado de ambos flujos, el de entrada y el de salida.

9. Para realizar la ejecución del programa teclee:

make ejecuta09

para la ejecución se esta forma se invoca de la siguiente forma:

java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio09 a A

```
/*
 *  @(#)Practica8Ejercicio09.java
 *
 *  Fundamentos de Programacion II. GITT.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 */

/**
 * Descripcion:
 *
 * version 1.0 Mayo 2014
 * Fundamentos de Programacion II
 */

package fp2.poo.practica8;

import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

/**
 * Programa que copia su entrada en su salida,
 * transformando un valor particular en otro.
 * Con valores proporcionados desde linea de comandos.
 */
public class Practica8Ejercicio09 {
    public static void main( String args[] ) {

        try{
            byte desde = (byte)args[0].charAt(0);
            byte hacia = (byte)args[1].charAt(0);
            int b = 0;

            while( (b = System.in.read()) != -1)
                System.out.write( (b == desde) ? hacia : b);

            }catch(IOException e){
                System.out.println(e);
            }catch(IndexOutOfBoundsException e){
                System.out.println(e);
            }
    }
}
```

En este ejemplo se toman el primer carácter de los dos argumentos en línea de comando. El ejemplo realiza un cambio de todas las ocurrencias del primer carácter y las transforma en el del otro carácter.

10. Para realizar la ejecución del programa teclee:

make ejecuta10

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

/**
 * Un flujo Pushback nos permite enviar de vuelta
 * cuando la aplicación lo requiera.
 * Pushback es generalmente útil para dividir la entrada en tokens.
 * Por ejemplo, los exploradores léxicos a menudo sólo saben que un token
 * ha terminado (como por ejemplo un identificador) cuando leen el primer
 * carácter que lo sigue. Una vez visto ese carácter, el explorador debe
 * devolverlo al flujo de entrada, para que quede disponible como el
 * primer carácter del siguiente token.
 *
 * Este ejercicio usa un flujo PushbackInputStream para indicar la secuencia
 * consecutiva más larga con el mismo byte de entrada.
 * Los resultados los muestra por pantalla.
 */
import java.io.IOException;
import java.io.PushbackInputStream;

public class Practica8Ejercicio10 {
    public static void main(String[] args) throws IOException{
        PushbackInputStream in = new PushbackInputStream (System.in);
        int max = 0; // secuencia más larga encontrada
        int maxB= -1; // el byte de esa secuencia
        int b; // byte actual de la entrada

        do{
            int cnt;
            int b1 = in.read(); // primer byte de la secuencia
            for(cnt = 1; (b = in.read()) == b1 ; cnt++)
                ;
            if(cnt > max){
                max = cnt; //recuerda la longitud
                maxB= b1; //recuerda el valor del byte
            }
            in.unread(b); //devuelve el inicio de la siguiente sec.
        }while(b != -1); //hasta el final de la entrada.
        System.out.println(max + " bytes de " + (char)maxB);
    }
}
```

Este programa lee la secuencia de caracteres idénticos más larga de un flujo. Usa el método `unread`, de la clase `PushbackInputStream`, para devolver datos al flujo.

11. Para realizar la ejecución del programa teclee:

make ejecuta11SinFichero

o bien

make ejecuta11ConFichero

Para realizar la lectura de teclado o bien del fichero.

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.InputStreamReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.Reader;

public class Practica8Ejercicio11 {
    public static void main(String args[]){
        Reader in = null;
        int cr = 0;
        int total = 0;
        int espacio = 0;
        try{
            if( args.length == 0){
                in = new InputStreamReader(System.in);
            }else{
                in = new FileReader(args[0]);
            }
            for(total = 0; (cr = in.read()) != -1 ; total++){
                if(Character.isWhitespace((char)cr))
                    espacio++;
            }
            in.close();
            System.out.println(total + " caracteres, " + espacio + " espacios");
        }catch(FileNotFoundException e){
            System.out.println(e);
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

Realiza la lectura de ficheros mediante la clase `FileReader`.

12. Para realizar la ejecución del programa teclee:

make ejecuta12

esta invoca al programa de la siguiente forma

```
java -classpath ./bin fp2.poo.practica8.Practica8Ejercicio12 ./ficherosSalida/FicheroSalidaPrueba12.txt
```

```
package fp2.poo.practica8;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.FileWriter;
import java.io.FileNotFoundException;
import java.io.Writer;

public class Practica8Ejercicio12 {
    public static void main(String args[]){
        InputStream     inp  = null;
        InputStreamReader in  = null;
        Writer         out  = null;
        int            cr   = 0;

        try{
            inp  = System.in;
            out = new FileWriter( args[0] );
            in  = new InputStreamReader(System.in);

            for( ; (cr = in.read()) != -1 ; ){
                out.write((byte)cr);
                System.out.print("") + (char)cr;
            }
            out.close();
            in.close();
        } catch ( IOException e ) {
            e.printStackTrace();
        }
    }
}
```

En este programa se realiza la escritura en un fichero de tipo **FileWriter**.

Trabajo a entregar

Implemente la clase **TrabajoAEntregar** que en el paquete **fp2.poo.practica8.XXX** (siendo XXX el uvus).

En la clase **TrabajoAEntregar**, se debe realizar la lectura de datos con flujos del tipo binario desde la entrada estándar. La lectura finalizará cuando se encuentra con el carácter ‘q’. Por pantalla se muestra el número total de caracteres leídos y el número de espacios en blanco introducidos desde la entrada estándar. Para comprobar si la información proporcionada desde el teclado es un espacio en blanco se utiliza el método estático **isWhitespace** de la clase **Character**.

Comprima el código en un archivo de nombre el uvus del alumno y entréguelo en la Actividad de la práctica.



PRACTICA 9: Tipos genéricos.

1. OBJETIVO

El objetivo de esta práctica es entender la utilidad de los tipos genéricos en java y su utilización en la elaboración y reutilización de código, así como las restricciones que presenta su utilización.

2. TIPOS GENÉRICOS

Un tipo genérico es una clase o interfaz genérica que está parametrizada con tipos. Los tipos genéricos permiten que una clase o método pueda operar con objetos de tipos diferentes, proporcionando seguridad en los tipos en tiempo de compilación. Evitan el uso del operador `cast`. Una posible solución para poder operar con objetos de tipos diferentes, sin utilizar tipos genéricos, es utilizar una referencia de la clase `Object`, tal y como se muestra a continuación.

Fichero: `Bolsa.java`

```
package fp2.poo.practica09;
public class Bolsa{
    private Object objeto;
    public void agrega(Object objeto) {
        this.objeto = objeto;
    }
    public Object obtiene() {
        return objeto;
    }
}
```

Fichero: `Ejemplo01.java`

```
package fp2.poo.practica09;

public class Ejemplo01 {
    public static void main(String[] args){
        // SOLO objetos Integer en Bolsa!
        Bolsa cajaDeInteger = new Bolsa();
        cajaDeInteger.agrega(new Integer(10)); //Linea a comentar
        //cajaDeInteger.agrega(new String("10"));
        /* Con el cast */
        Integer unInteger = (Integer)cajaDeInteger.obtiene();
        /* Sin el cast */
        //Integer unInteger = cajaDeInteger.obtiene();
        System.out.println(unInteger);
    }
}
```

En la clase `Ejemplo01` se muestra el uso de la clase `Bolsa`, que contiene el atributo `objeto` de tipo `Object`. Usa el método `agrega`, para proporcionar un objeto de

tipo `Integer` que se guarda en el atributo `objeto`. También usa el método `obtiene`, para obtener el objeto.

Descargue el código de la plataforma para esta práctica y descomprímalo. Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba01

Comente la línea que usa el objeto de tipo `Integer(cajaDeInteger.agrega(new Integer(10)); //Linea a comentar)` y descomente la línea siguiente (`cajaDeInteger.agrega(new String("10"));`). Vuelva a compilar y ejecutar el código. En este caso, se generará una excepción en tiempo de ejecución. Compruebe que si no utiliza el cast genera un error en compilación.

Un tipo de dato genérico es una clase, o interfaz, que está parametrizada con los tipos de datos que se desea admitir, siempre que estos no sean tipos de datos primitivos. A los tipos de datos genéricos también se les denominan **tipos parametrizados**. Para declarar una clase o interfaz de tipo genérico (también llamadas clase o interfaz genérica) debemos utilizar el operador diamante (`<·>`) en cuyo interior se utiliza al menos una variable genérica (`T, U, V, K, S, etc...`) que será sustituida en tiempo de compilación por el tipo de dato especificado en su declaración. En el siguiente ejemplo se utiliza una clase genérica llamada `Caja`.

```
Fichero: Caja.java
package fp2.poo.practica09;

/**
 * Version con Genericos de la clase Caja.
 */
public class Caja<T> {

    private T objeto;

    /** Constructor sin parámetros.
     * El constructor está sobrecargado.
     */
    public Caja( ){
        this.objeto = null;
    }

    /** Constructor con un parámetro.
     * El constructor está sobrecargado.
     */
    public Caja( T objeto ){
        this.objeto = objeto;
    }

    /** Metodo para añadir un objeto. */
    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    /** Metodo para obtener el objeto. */
    public T obtiene() {
        return objeto;
    }
}
```

```

Fichero: Ejemplo02.java
package fp2.poo.practica09;

public class Ejemplo02 {
    public static void main ( String[] args ) {
        /* Se crea un objeto Caja especificando como el tipo "String" */
        Caja<String> caja01DeString = null; /* Declaracion de la variable*/
        caja01DeString = new Caja<String>(); /* Construcción del obj. */
        caja01DeString.agrega("Fundamentos de Programación II"); /* Uso.*/
        System.out.println(caja01DeString.obtiene());

        Caja<String> caja02DeString = null;
        caja02DeString = new Caja<String>("Fundamentos de Programación II");
        System.out.println(caja02DeString.obtiene());

        /* Se crea un objeto Caja especificando como el tipo "Integer" */
        Integer unEntero = null;
        unEntero = new Integer(2);
        Caja<Integer> cajaDeInteger = new Caja<Integer>();
        cajaDeInteger.agrega(unEntero);
        System.out.println("Fundamentos de Programación "
                           +cajaDeInteger.obtiene());
    }
}

```

En la clase **Ejemplo02.java** se crean objetos de la clase **Caja** indicando los tipos **String** e **Integer**.

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba02

Observe que en este caso no es necesario el cast al obtener el objeto de **Caja**. Esto es debido a que los tipos genéricos evitan el uso de los operadores casts, que realizan una conversión explícita al tipo nombrado en el operador cast, ya que el compilador sabe el tipo de dato que se está admitiendo, permitiendo que las clases implementen algoritmos genéricos. Observe que uno de los constructores de la clase **Caja** admite un parámetro de tipo **T**.

Los tipos genéricos añaden estabilidad al código permitiendo la detección de fallos en tiempo de compilación así, por ejemplo, si tenemos en el código una caja genérica, **Caja<T>**, podemos conseguir que admita datos de tipo cadena (**Caja<String>**), de tipo entero (**Caja<Integer>**) o un booleano (**Caja<Boolean>**), sin embargo, si intentamos añadir a un objeto de la clase **Caja<String>** (que admite un tipo de dato cadena) otro tipo de dato (como puede ser un número entero), en tiempo de compilación dará un error debido a que son tipos de datos incompatibles.

En el siguiente ejemplo se ha declarado un objeto del tipo **Caja** parametrizado con el tipo **Integer**. Al compilar **Ejemplo03.java**, se produce un error debido a que se intenta invocar el método **agrega** de **Caja** con un **String**.

Fichero: Ejemplo03.java

```

package fp2.poo.practica09;

public class Ejemplo03 {
    public static void main(String[] args) {

        // SOLO objetos Integer en Caja!
        Caja<Integer> cajaInteger = null;
        cajaInteger = new Caja<Integer>();

        // Se intenta agregar un objeto de tipo String
        cajaInteger.agrega("10"); // es un String

        //cajaInteger.agrega(new Integer(10));
        System.out.println(cajaInteger.obtiene());
    }
}

```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba03

Cambie el programa para que se compile y ejecute correctamente.

Los errores que ocurren en tiempo de compilación son más rápidos de detectar y corregir que los que ocurren en tiempo de ejecución, por eso los tipos parametrizados ofrecen más seguridad (que la utilización de la referencia a Object).

Dado que estamos trabajando con parámetros, también es posible utilizar más de un tipo para parametrizar una clase genérica.

Fichero: Pareja.java

```

package fp2.poo.practica09;

public class Pareja<X, Y> {
    private X primero;
    private Y segundo;

    public Pareja(X a1, Y a2) {
        this.primero = a1;
        this.segundo = a2;
    }
    public X getPrimero() {
        return this.primero;
    }
    public Y getSegundo() {
        return this.segundo;
    }
    public void setPrimero(X arg) {
        this.primero = arg;
    }
    public void setSegundo(Y arg) {
        this.segundo = arg;
    }
}

```

Fichero: Ejemplo04.java

```

package fp2.poo.practica09;

public class Ejemplo04{
    public static void main (String args[]){
        Pareja<String, Integer> par
            = new Pareja<String, Integer>("cadena", new Integer(1));
        System.out.println("Parametro primero: " + par.getPrimero());
        System.out.println("Parametro segundo: " + par.getSegundo());
        par.setPrimero("cambio cadena");
        par.setSegundo(new Integer(21));
        System.out.println("Parametro primero: " + par.getPrimero());
        System.out.println("Parametro segundo: " + par.getSegundo());
    }
}

```

En este ejemplo se muestra que a la clase **Pareja** se le proporcionan dos tipos representados mediante X e Y. La clase **Pareja** se utiliza en el método **main** de la clase **Ejemplo04**.

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba04

Los tipos genéricos se pueden aplicar tanto a métodos como a constructores. Un ejemplo se puede ver en **Caja05.java**.

Fichero: Caja05.java

```

package fp2.poo.practica09;

public class Caja05 {

    public Caja05 () {
        System.out.println("Constructor Caja05 sin parametros");
    }

    public <T> Caja05 ( T tipo ) {
        System.out.println("\nConstructor Caja05 con parametros");
        System.out.println("Tipo y su valor: " + tipo.getClass().getName()
                           +"\t"+ tipo +"\n");
    }

    public <U> void metodoConGenericos( U parametro ){
        System.out.println("Tipo y su valor: "
                           + parametro.getClass().getName() +"\t"+ parametro);
    }

    public static <M> void metodoConGenericosEstatico( M variable ){
        System.out.println("Tipo y su valor: "
                           + variable.getClass().getName() +"\t"+ variable);
    }
}

```

Fichero: Ejemplo05.java

```

package fp2.poo.practica09;

public class Ejemplo05 {
    public static void main( String args[] ){
        Caja05 ref01 = new Caja05();
        Caja05 ref02 = new Caja05(new Long(10L));
        Caja05 ref03 = new <Long>Caja05(new Long(10L));

        System.out.println("Con metodo no estático.\n");
        ref01.<String> metodoConGenericos( "Fundamentos de Programacion II");
        ref01.<Integer>metodoConGenericos( new Integer( 10 ) );
        ref01.<Double> metodoConGenericos( new Double ( 10.0 ) );

        System.out.println();
        System.out.println("Con metodo estático.\n");

        Caja05.<String> metodoConGenericosEstatico( "Fundamentos de Programacion II");
        Caja05.<Integer>metodoConGenericosEstatico( new Integer( 10 ) );
        Caja05.<Double> metodoConGenericosEstatico( new Double ( 10.0 ) );
    }
}

```

El primer aspecto a destacar en este ejemplo es que la clase `Caja05` no contiene parámetro de tipo (es decir, no aparece como `Caja05<T>`) sino que los tipos son proporcionados a los métodos (estáticos o no estáticos) y a uno de los dos constructores de la clase para poder proporcionar un parámetro local.

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba05

La forma de crear un objeto de la clase **Caja05**, es invocando a alguno de sus constructores.

Para invocar al primer constructor (**Caja05()**) no se proporciona ningún parámetro (**Caja05 ref01 = new Caja05();**).

Para invocar al segundo constructor (**public <T> Caja05 (T tipo)**) se proporciona el tipo después de **new** y antes de invocar al constructor (es decir, **Caja05 ref02 = new <Long>Caja05(new Long(10L));**), esta operación se puede realizar también sin indicar el tipo después de **new** (**Caja05 ref02 = new Caja05(new Long(10L));**)

En este ejemplo se presenta además la invocación a dos métodos uno estático y otro no estático. En el caso del método no estático la invocación al método se realiza mediante la referencia al objeto proporcionando el tipo antes de indicar el método. En el caso del método estático se proporciona el nombre de la clase en lugar de la referencia al objeto.

Algunas restricciones al uso de genéricos

Existen algunas restricciones al uso de tipos genéricos y parte de ellas las mostramos en estos ejemplos. Para el uso efectivo de los tipos genéricos en Java, debemos tener en cuenta las siguientes consideraciones:

- No se pueden instanciar tipos genéricos con tipos de datos primitivos (**int**, **long**, **boolean**, **short**, **float**, **double** y **char**) en su lugar podemos utilizar sus correspondientes envolventes: **Integer**, **Long**, **Boolean**, **Short**, **Float**, **Double** y **Character**.

```
Fichero: Caja06.java
package fp2.poo.practica09;

public class Caja06<T>{

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

```
Fichero: Ejemplo06.java
package fp2.poo.practica09;

class Ejemplo06 {
    public static void main(String[] args) {
        // El uso de genéricos con tipos de datos simples
        // no está permitido.
        // Estos ejemplos generarán error en compilación.

        Caja06<int>    caja01  = new Caja06<int>();
        Caja06<long>   caja02 = new Caja06<long>();
        Caja06<float>  caja03 = new Caja06<float>();
        Caja06<double> caja04 = new Caja06<double>();

        // Estos ejemplos NO generarán error en compilación.
        Caja06<Integer> caja01  = new Caja06<Integer>();
        Caja06<Long>    caja02 = new Caja06<Long>();
        Caja06<Float>   caja03 = new Caja06<Float>();
        Caja06<Double>  caja04 = new Caja06<Double>();

        caja01.add(new Integer (1));
        caja02.add(new Long    (2L));
        caja03.add(new Float   (3.f));
        caja04.add(new Double  (4.d));

        System.out.println(caja01.get());
        System.out.println(caja02.get());
        System.out.println(caja03.get());
        System.out.println(caja04.get());
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba06

Compruebe que el ejemplo genera error en compilación debido a que se están utilizando los tipos de datos simples. Comente las líneas de código que generan error. Recompile y ejecute.

- No se puede crear una instancia a partir de un tipo proporcionado como parámetro. Por tanto el siguiente ejemplo genera un error.

```
Fichero: Ejemplo07.java
package fp2.poo.practica09;

public class Ejemplo07 {
    public static void main(String[] args) {
        Ejemplo07.metodo();
    }
    public static <E> void metodo() {
        E elem = new E(); // error en compilacion
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba07

- No se puede declarar campos estáticos cuyos tipos son de tipo genérico.

```
Fichero: Caja08.java
package fp2.poo.practica09;

public class Caja08<T>{

    private static T atributoPrivado;      //ERROR
    private T objeto;

    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    public T obtiene() {
        return objeto;
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba08

- No puede usarse el operador **instanceof** con tipos genéricos. Esto se debe a que el compilador de Java borra todos los tipos genéricos, y no se puede verificar qué tipo parametrizado se está utilizando al ejecutarse el código. Por tanto el siguiente código generará error en compilación.

```
Fichero: Caja09.java
package fp2.poo.practica09;

public class Caja09<T>{
    private T objeto;

    public Caja09( ){
        this.objeto = null;
    }

    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    public T obtiene() {
        return objeto;
    }
}
```

```
Fichero: Ejemplo09.java
package fp2.poo.practica09;

public class Ejemplo09 {
    public static void main(String[] args)throws Exception  {
        Caja09<Integer> ref = new Caja09<Integer>();
        if(ref instanceof Caja09<Integer> ) {
            System.out.println("Error en compilacion...");
        }
        //if(ref instanceof Caja09<?> ) {
        //    System.out.println("No hay error en compilacion...");
        //}
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba09

En esta prueba generará un error en compilación al usar el operador **instanceof**.

No se puede saber a priori que la variable genérica **T** sea en el futuro un **Integer**. En su lugar podemos utilizar las **wildcards**. Comente la sentencia **if** que genera el error y descomente el **if** que se proporciona comentado, y pruebe su compilación.

- No pueden crearse matrices a partir de tipos parametrizados.

```
Fichero: Caja10.java
package fp2.poo.practica09;

public class Caja10<T>{

    private T objeto;

    public Caja10( ){
        this.objeto = null;
    }

    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    public T obtiene() {
        return objeto;
    }
}
```

```
Fichero: Ejemplo10.java
package fp2.poo.practica09;

public class Ejemplo10 {
    public static void main(String[] args)throws Exception  {
        Caja10<Integer> ref [] = new Caja10<Integer>()[10];
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba10

Al compilarlo genera un error debido a que no se pueden crear matrices a partir de tipos parametrizados.

- No se puede crear (**new**), capturar (**catch**) o lanzar (**throw**) objetos de excepción de tipos parametrizados.

Una clase genérica no puede heredar de la clase **Throwable**, directa o indirectamente.

```
Fichero: ExcepcionConGenericos.java
package fp2.poo.practica09;

class ExcepcionConGenericos<T> extends Exception {
    // error en tiempo de compilacion
    /* ... */
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba11

```
Fichero: ThrowableConGenericos.java
package fp2.poo.practica09;

public class ThrowableConGenericos<T> extends Throwable {
    /* ... */
    // error en tiempo de compilacion
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba12

En ambos casos generan error en compilación.

Sin embargo puede usarse un parámetro genérico en la sentencia **throws** del método:

```
Fichero: Caja13.java
package fp2.poo.practica09;

class Caja13<T extends Exception> {
    private Integer matriz [];

    public void get() throws T {
        // Hacer algo...
        matriz[0]= 1;
    }
}
```

```

Fichero: Ejemplo13.java
package fp2.poo.practica09;

public class Ejemplo13 {
    public static void main(String[] string) {
        Caja13<NullPointerException> obj
            = new Caja13<NullPointerException>();
        try {
            obj.get();
        } catch (NullPointerException e) {
            //Gestion de la excepcion
            System.out.println("Excepcion capturada");
        }
    }
}

```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba13

En este ejemplo la excepción se lanza en el método `get` de la clase `Caja13<T extends Exception>`, y se captura en el método `main`. La excepción se lanza ya que la matriz no está instanciada y se accede a la posición 0.

- No se pueden sobrecargar métodos que usan tipos parametrizados, esto es debido al borrado que se produce de la variable genérica. Suponiendo que la variable genérica es `T` ¿por qué tipo se sustituye? ¿Por **String** o **Integer**? No puede ser de los dos tipos.

```

Fichero: Caja14.java
package fp2.poo.practica09;

public class Caja14<T> {
    private T objeto;

    public Caja14( T objeto ){
        this.objeto = objeto;
    }

    public void agrega(T objeto) {
        this.objeto = objeto;
    }

    public T obtiene() {
        return objeto;
    }
}

```

Fichero: Ejemplo14.java

```
package fp2.poo.practica09;

class Utilidades {
    public static void imprime(Caja14<Integer> c){
        System.out.println(c.obtiene());
    }

    //public static void imprime(Caja14<String> c){
    //    System.out.println(c.obtiene());
    //}
}

public class Ejemplo14 {
    public static void main(String[] string) {
        Caja14<Integer> obj = new Caja14<Integer>(new Integer(7));
        Utilidades.imprime( obj );
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba14

Para probar la sobrecarga de método quite los comentarios al código y vuelva a ejecutarlo. Comprobará que se genera un error en compilación.

Ejercicio 1. Cree una clase genérica denominada “**ExpendedoraGenerica**” para ello:

- Cree tres atributos privados. Uno de los atributos de tipo **Random** (**java.util.Random**), otro de tipo int que se usa como contador de las inserciones que se realizan en una matriz, y el tercero la matriz (**T listaElementos[]**). En el constructor de la clase se proporciona la matriz instanciada.
- Implemente un método genérico “**agregarElemento**” (**public void agregarElemento(T elemento)**) que permita añadir un nuevo elemento a la lista.
- Implemente un método llamado “**aleatorio**” (**public T aleatorio()**) que devuelva un objeto de la lista de forma aleatoria. Utilice el método **nextInt** (clase **Random**) y ayúdese del tamaño de la lista para acotar el generador de números aleatorios.
- Cree una clase principal llamada “**Main**” y pruebe en ella que la expendedora genérica devuelve un elemento de forma aleatoria.

Wildcard

En un código genérico de Java, el signo de interrogación (?), representa un tipo desconocido. Los “**wildcard**” o comodines pueden ser utilizados en varias situaciones: como tipo de un parámetro, un atributo o una variable local; En ocasiones como un tipo de retorno.

```
Fichero: Caja15.java
package fp2.poo.practica09;

public class Caja15<T>{
    /** Variable genérica */
    private T t;

    /** Constructor */
    public Caja15(T t){
        this.t = t;
        //System.out.println("T: " + t.getClass().getName());
    }

    /** Método que asigna un nuevo objeto */
    public void agrega(T t) {
        this.t = t;
    }

    /** Método que obtiene el objeto almacenado */
    public T obtiene() {
        return t;
    }
}
```

```
Fichero: Ejemplo15.java
package fp2.poo.practica09;

class Ejemplo15 {
    public static void main(String[] string) {
        Caja15<?> obj = new Caja15<Integer>(new Integer(7));
        System.out.println(obj.obtiene());
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba15

En este ejemplo se usa una referencia del tipo **Caja15<?>**, y se utiliza para obtener el objeto mediante el método **obtiene()**.

LIMITAR LOS TIPOS PARAMETRIZADOS

En ocasiones se desea limitar los tipos parametrizados que pueden ser usados como argumentos en un método. Por ejemplo, si un método opera con números podríamos aceptar únicamente instancias del tipo **Number** y sus subclases, estamos, por lo tanto, especificando un tipo paramétrizado limitado. Las limitaciones pueden ser de dos tipos:

- **Limitación superior:** Se denota por `<? extends T>` y permite el tipo **T** y a los tipos que herede de **T** (todos sus descendientes).
- **Limitación inferior:** Se denota por `<? super T>` y permite el tipo **T** y los tipos que son superclase de **T** (todos sus ascendentes).

```
Fichero: Ejemplo16.java
package fp2.poo.practica09;
public class Caja16<T>{
    /** Variable genérica */
    private T t;

    /** Constructor */
    public Caja16(){
        this.t = null;
    }

    /** Constructor */
    public Caja16(T t){
        this.t = t;
        //System.out.println("T: " + t.getClass().getName());
    }

    /** Método que asigna un nuevo objeto */
    public void agrega(T t) {
        this.t = t;
    }

    /** Método que obtiene el objeto almacenado */
    public T obtiene() {
        return t;
    }
}
```

```
Fichero: Ejemplo16.java
package fp2.poo.practica09;

public class Ejemplo16 {
    public static void main (String args[]){
        Caja16<Integer> obj1 = new Caja16<Integer>();
        Caja16<Integer> obj2 = new Caja16<Integer>(new Integer(12));
        System.out.println("Valor :" + obj1.obtiene());
        System.out.println("Valor :" + obj2. obtiene ());
        obj1.agrega(new Integer(15));
        System.out.println("Valor :" + obj1. obtiene ());

        Caja16<? extends Number> obj3 = new Caja16<Long>(new Long(10L));
        Caja16<? super String> obj4 = new Caja16<String>(new String());
        obj3 = obj1;
        //obj3 = obj4; //ERROR
    }
}
```

Para ver el funcionamiento del programa ejecute la siguiente orden:

make prueba16

Descomente la línea //obj3 = obj4; //ERROR

Y compruebe que se genera un error al ejecutar el programa.

Borrado de tipos en la compilación

Los genéricos en el lenguaje Java proporciona comprobaciones más estrictas a la hora de compilar y permite soportar la programación genérica. Para implementar la programación genérica, el compilador de Java reemplaza todos los tipos paramétricos genéricos por los correspondientes objetos o tipos ligados. El bytecode producido, contiene únicamente clases, interfaces y métodos ordinarios, sin distinguir ningún tipo genérico.



PRÁCTICA 10: Colecciones

1. OBJETIVO

El objetivo de esta práctica es presentar los conceptos relacionados con colecciones en java. Descargue de la plataforma virtual el código ejemplo, necesario para realizar la práctica.

2. CÓDIGO PROPORCIONADO

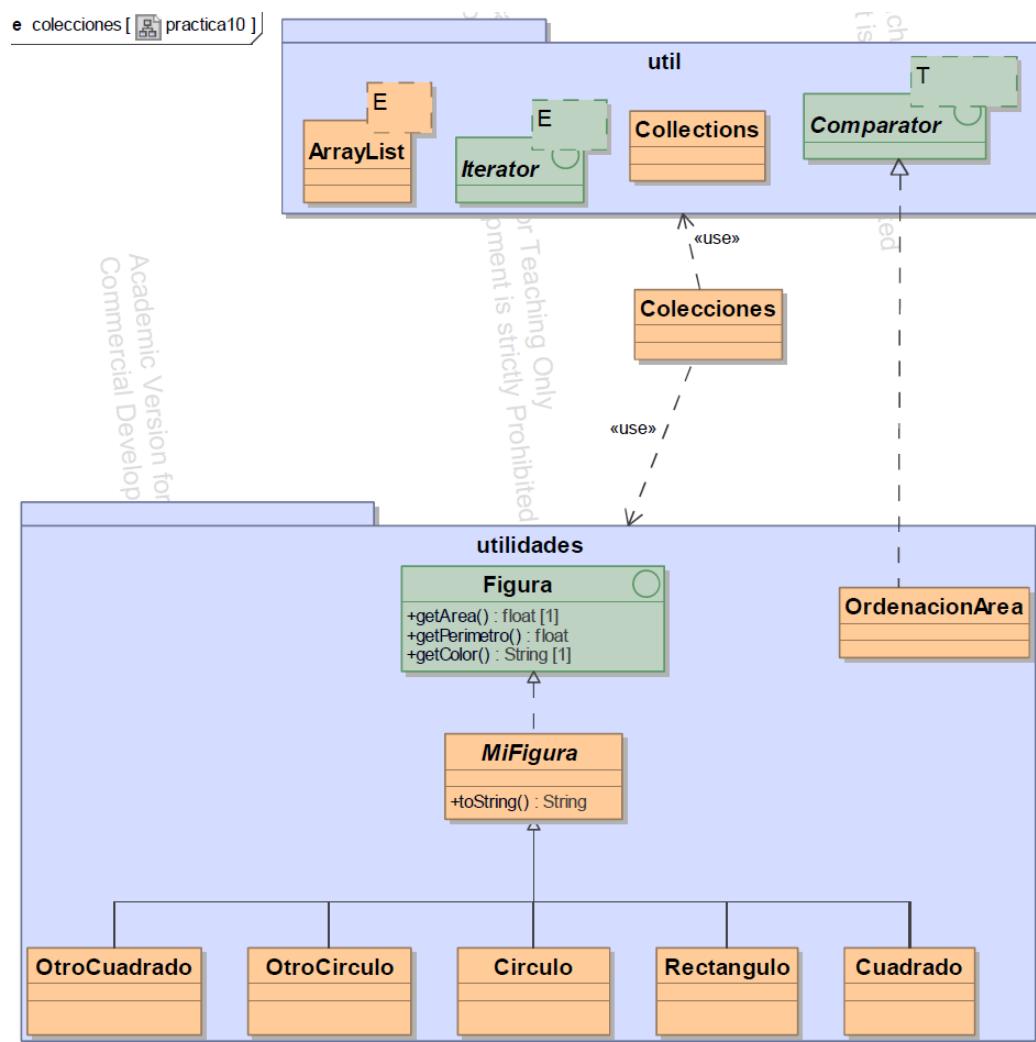


Figura 1. Diagrama de clase para la realización de la práctica.

El diagrama de clases del código proporcionado se puede observar en la Figura 1, que incluye además las clases utilizadas de Java correspondientes a las colecciones. En el fichero descargado, una vez descomprimido, se puede encontrar:

- 1) Fichero makefile para compilar y ejecutar el código.
- 2) Fichero makeJavadoc para generar la documentación en html.

```
make -f makeJavadoc
```

- 3) Carpeta “utilidades”, que contiene una serie de clases auxiliares para la realización de la práctica y que se detallan a continuación:
- Interfaz **Figura**: indica la firma de métodos para usar figuras geométricas genéricas, puede verla en Código 1.

Fichero: Figura.java
<pre>package fp2.poo.practica10.utilidades; public interface Figura { float getArea(); float getPerimetro(); String getColor(); }</pre>

Código 1. Definición de la interfaz **Figura**.

- Clase abstracta **MiFigura**. Observe en Error: No se encuentra la fuente de referencia como implementa **Figura** pero deja sus métodos declarados como abstractos, de modo que son sus subclases las que deben implementar los métodos de **Figura**. Se consigue que todas las implementaciones de **Figura** que hereden de **MiFigura** tengan sobreescrito el método **toString** (de la clase **Object**) para convertir los objetos figura a un **String**. Esto facilita la depuración y análisis del código ejemplo, ya que al imprimir por pantalla el objeto se muestra el **String** que devuelve este método. **getClass** devuelve un objeto de tipo **Class** (que se corresponde con la clase del objeto objetivo), el método **getName** de la clase **Class** devuelve como un **String** el nombre de la clase, incluyendo la ruta completa de paquetes. Observe el mecanismo que se utiliza para imprimir por pantalla únicamente el nombre de la clase (sin incluir los paquetes), se recurre a los métodos de **String**; **lastIndexOf** y **substring**, analice su comportamiento, consulte la clase **String** en la documentación de Oracle si lo necesita [1].

Fichero: MiFigura.java
<pre>package fp2.poo.practica10.utilidades; public abstract class MiFigura implements Figura { public String toString(){ int indicepunto=this.getClass().getName().lastIndexOf("."); return this.getClass().getName().substring(indicepunto+1) +" de color "+this.getColor()+"\n"; } abstract public float getArea(); abstract public float getPerimetro(); abstract public String getColor(); }</pre>

Código 2. Definición de la clase abstracta *Mifigura* sobreescribe el método **toString** de la clase **Object** (no implementa los métodos de la interfaz **Figura**, los deja como abstractos, serán sus subclases las que los implementen).

- Subclases de la clase abstracta *Mifigura*: varias subclases de la clase anterior (*Mifigura*) para representar distintos tipos de figuras geométricas

(Circulo, OtroCirculo, Cuadrado, OtroCuadrado, Rectangulo). Todas las subclase implementan los métodos de la interfaz Figura.

- OrdenacionArea: implementación de la interfaz Comparator [2] que permitirá comparar y ordenar figuras geométricas en función de su área. Observe la construcción de este comparador “personalizado” que compara figuras por su área en Código 3. Posteriormente verá cómo este comparador permitirá ordenarlas de mayor a menor (lo verá en Código 14). Este diseño de la librería permite la creación de algoritmos personalizados a los requisitos del programador facilitando además su reutilización. Observe en Código 3 el uso del método compareTo del envolvente Float (todos los envolventes lo incluyen ya que implementan la interfaz Comparable).

Fichero: OrdenacionArea.java
package fp2.poo.practica10.utilidades; ... public class OrdenacionArea implements Comparator<Figura>{ public int compare(Figura f1, Figura f2) { float a1=f1.getArea(); float a2=f2.getArea(); Float area1=new Float(a1); Float area2=new Float(a2); return area2.compareTo(area1); } }

Código 3. Implementación de un comparador de figuras personalizado. Clase OrdenacionArea

- 4) Carpeta “colecciones”, que contiene la clase principal, en la que se usa la clase ArrayList [3], como ejemplo de colección.

3. INTRODUCCIÓN A LAS COLECCIONES

Una **colección** es un objeto que agrupa múltiples **elementos** en una sola unidad, por ello a veces se conoce también como contenedor, puede verse como una agregación de objetos (ya que cada **elemento** es a su vez un objeto). El **entendimiento y manejo de colecciones está muy relacionado con los conceptos de estructuras dinámicas estudiados en la asignatura “Fundamentos de programación I”**.

El paquete java.util incluye el entorno de trabajo (Framework) denominado “Collections” [4,5] (Java Collections Framework), que proporciona clases que permiten organizar y manipular colecciones de objetos, de cualquier tipo, de una forma unificada y estandarizada. Esta API proporciona interfaces para gestionar estructuras dinámicas, independientemente de la clase a la que pertenezcan los objetos de la estructura, así como implementaciones de las mismas (listas para usar). De este modo, si se reutilizan las clases ya proporcionadas, el esfuerzo de programación para manejar colecciones de objetos se reduce, ya que estas clases nos simplifican mucho la manipulación de las mismas (abstraen de la gestión de memoria, proporcionan métodos para la inserción y recuperación de los **elementos**, proporcionan algoritmos útiles, como los de ordenación...), optimizando además la eficiencia de los programas. Y por otro lado, si se necesita crear nuevas funcionalidades para la gestión de colecciones, más personalizadas a nuestras necesidades (nuevos algoritmos de

ordenación, métodos de búsqueda...) se facilita que otros puedan reutilizar nuestro código siempre que se implemente conforme a las interfaces propuestas.

De modo que resumiendo, el hecho de tener este entorno estandarizado facilita la interoperatividad y reutilización, reduciendo además el esfuerzo de diseño, de desarrollo y de aprendizaje de nuevas interfaces de programación de aplicaciones (API).

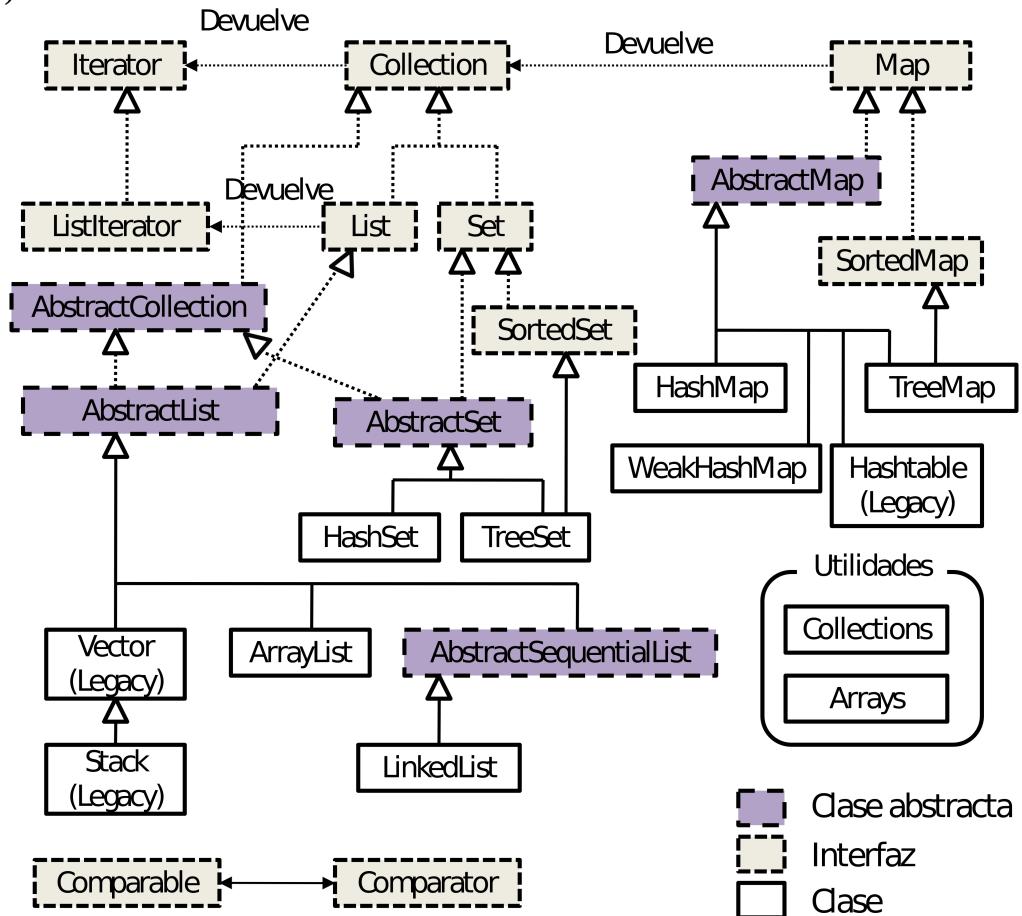


Figura 2. Elementos principales de la API para la gestión de colecciones de objetos.

El entorno de trabajo, que podemos entender como una API y cuyos elementos principales puede verse en la Figura 2, incluye:

- 14 interfaces de colecciones, base para representar diferentes tipos de colecciones (conjuntos, listas, mapas...).
- Implementaciones (realizaciones) básicas de las interfaces.
- Implementaciones abstractas, parciales, que facilitan a los programadores la realización de sus propias implementaciones.
- Implementaciones heredadas de versiones previas de Java (como **Vector** y **Hashtable**).
- Implementaciones de propósito específico, para situaciones particulares o con características optimizadas (uso concurrente, sincronización, alto rendimiento...).
- Implementaciones diseñadas especialmente para facilitar el uso concurrente.
- Implementaciones de tipo “envoltorio”, que añaden funcionalidades a las más básicas.

- Algoritmos, en forma de métodos estáticos que proporcionan funciones útiles y reutilizables para la gestión de colecciones (como por ejemplo algoritmos de ordenación de listas). Son polimórficos, el mismo método puede utilizarse en diferentes implementaciones de interfaz.
- Interfaces de apoyo para las de colecciones.
- Funciones útiles para el manejo de arreglos de primitivas y referencias de objetos.

La interfaz principal es `Collection<E>` [6], de ella heredan `Set`, `List`, `SortedSet`, `NavigableSet`, `Queue`, `Deque`, `BlockingQueue` y `BlockingDeque`.

Las otras interfaces `Map`, `SortedMap`, `NavigableMap`, `ConcurrentMap` and `ConcurrentNavigableMap` no heredan de `Collection`, ya que representan “mapas” (estructuras de pares clave/valor con las que trabajará la próxima práctica) en lugar de auténticas colecciones, aunque sí contienen operaciones que permiten manipularlas como si fueran colecciones.

Además de estas interfaces se incluyen implementaciones, como se ha comentado anteriormente. La tabla que se muestra a continuación incluye sólo las clases más utilizadas y generales.

		Implementaciones				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Tabla 1. Principales interfaces y realizaciones del entorno

4. LA INTERFAZ COLLECTION<E> Y SUS DERIVADAS

`Collection<E>` [6] es una interfaz genérica, de modo que el tipo concreto de objeto contenido se especifica cuando se declara la instancia de una implementación de esta interfaz, o más bien de alguna de sus subinterfaces más específicas (como `Set` o `List`), ya que no se proporcionan implementaciones directas de `Collection`. Puede ver un ejemplo de declaración en el código proporcionado (clase principal, `Colecciones`). En este caso se utiliza la implementación `ArrayList` de la interfaz `List`, especificando que los elementos de la colección serán implementaciones de la interfaz `Figura`. Esto se muestra en el siguiente ejemplo de código:

```
List<Figura> serieDeFiguras = new ArrayList<Figura>();
```

Cada subinterfaz de `Collection` especifica unas características determinadas en la gestión de sus elementos (ordenación, elementos duplicados...), y por tanto **es importante elegir la implementación que más se adapte a las necesidades del desarrollo**.

La interfaz `Collection` declara métodos básicos de gestión de una colección, por ejemplo:

- `int size()` o `boolean isEmpty()`: para conocer aspectos de su tamaño. Puede ver ejemplos de su uso en el código proporcionado en:

```
if(serieDeFiguras.isEmpty())
    System.out.println("NO hay elementos en la colección");
else
    System.out.println("SI hay elementos en la colección, concretamente "
        +serieDeFiguras.size()+"\n");
```

- `boolean add(E element)` o `boolean remove(Object element)`: para añadir o eliminar un elemento de la colección. Puede ver [ejemplos de su uso en el código proporcionado](#), en:

```
Figura cuad3 = new OtroCuadrado (8.9f, "SERENITY");
serieDeFiguras.add (cuad3);
serieDeFiguras.remove(cuad3);
```

- `boolean contains(Object element)`: para verificar si `element` es un elemento de la colección. Puede ver ejemplo de su uso en el código proporcionado en:

```
if(serieDeFiguras.contains(cuad3))
    System.out.println(cuad3
        + " SI pertenece a la colección y esta en la posición "
        + serieDeFiguras.indexOf(cuad3)
        + "\n\n");
else
    System.out.println(cuad3+" NO pertenece a la colección\n\n");
```

- `Iterator<E> iterator()`: para proporcionar un iterador que permite moverse por la colección. Puede ver un ejemplo de obtención y uso de un iterador en el ejemplo proporcionado en:

```
Iterator<Figura> iterador=serieDeFiguras.iterator();
while(iterador.hasNext()){
    System.out.println(iterador.next());
}
```

La clase `Collection` incluye además operaciones que operan en la colección completa. Lanzan una excepción del tipo `NullPointerException` si la colección o el objeto que se les proporciona son nulos (`null`). Algunos ejemplos son:

- `boolean containsAll(Collection<?> c)`: devuelve `true` si la colección objetivo contiene todos los elementos en la colección especificada como argumento.
- `boolean addAll(Collection<? extends E> c)`: añade todos los elementos de la colección especificada como argumento en la objetivo.
- `boolean removeAll(Collection<?> c)`: elimina de la colección objetivo todos los elementos que estén también contenidos en la especificada como argumento.
- `boolean retainAll(Collection<?> c)`: elimina de la colección objetivo todos los elementos que no estén contenidos en la especificada como argumento. Así que mantiene sólo aquellos que estén contenidos en `c`.
- `void clear()`: elimina todos los elementos de la colección.

`addAll`, `removeAll`, y `retainAll` devuelven `true` si la colección objetivo se modificó al ejecutar el método.

También se incluyen métodos que proporcionan conversiones de colecciones a matrices, trasladando los elementos de una colección a una matriz. Principalmente se utilizan para hacer de puente con APIs que necesiten matrices como entrada.

- `Object[] toArray()`: los elementos de la colección objetivo se meten en un nuevo array de objetos con un tamaño idéntico al número de elementos de la misma. Ejemplo de invocación: `Object[] a = c.toArray();` (los elementos de la colección `c`, objetivo, se meten en `a`)
- `<T> T[] toArray(T[] a)`: los contenidos de la colección objetivo se trasladan a una nueva matriz con elementos de la clase `T`, cuya longitud es idéntica al número de elementos de la colección objetivo. Ejemplo de invocación: `String[] a = c.toArray(new String[0]);` (los elementos de la colección `c`, objetivo, se meten en `a`)

Por último, se proporcionan algoritmos genéricos de ordenación, búsqueda, etc, como se puede observar en el ejemplo en el que se utiliza un algoritmo, proporcionado por el método estático `reverse` de la clase `Collections[7]`, que permite cambiar el orden de los elementos de la colección, dándoles la vuelta:

```
System.out.println("Orden original \n"+serieDeFiguras);
Collections.reverse(serieDeFiguras);
System.out.println("\nNuevo Orden \n"+serieDeFiguras);
```

Y facilidades para que se puedan crear otros y reutilizarlos de forma sencilla y extensa. Como se puede observar en el ejemplo proporcionado, en el que se ha realizado una implementación de la clase `Comparator` (a la que hemos llamado `OrdenacionArea`) que permite comparar figuras por el tamaño de su área y que se reutiliza para ordenar las figuras de la colección en función de ésta utilizando el método estático `sort` de la clase `Collections`. Observe que se utiliza `compareTo`, método disponible en los envolventes de tipos genéricos, que permite comparar dos objetos de ese tipo. Esto es porque estos envolventes (wrappers) implementan la interfaz `Comparable`. Puede ver el código de ejemplo a continuación:

```
Fichero: OrdenacionArea.java
public class OrdenacionArea implements Comparator<Figura>{
    public int compare(Figura f1, Figura f2) {
        float a1=f1.getArea();
        float a2=f2.getArea();
        Float area1=new Float(a1);
        Float area2=new Float(a2);
        return area2.compareTo(area1);
    }
}
```

```
OrdenacionArea ordenador=new OrdenacionArea();
Collections.sort(serieDeFiguras,ordenador);

System.out.println(
    "\nAhora ordenadas por Area, de mayor a menor \n");

for (Figura tmp: serieDeFiguras) {
    System.out.println(tmp+": Area= "+tmp.getArea());
}
```

Las subinterfaces derivadas de `Collection` son:

- `Set`: una colección que no puede contener elementos duplicados. Se usa para representar conjuntos (las cartas de una baraja, las asignaturas en las que está matriculado un alumno...)
- `SortedSet`: es un tipo particular de conjunto en el que los elementos se gestionan en orden ascendente. Se incluyen algunas operaciones adicionales que sacan partido de esa ordenación. Se utilizan normalmente cuando los elementos del conjunto tienen esta naturaleza ordenada, por ejemplo un listado de palabras (orden alfabético) o los socios de un club (según su número de socio).
- `List`: una colección ordenada, o secuencia. Puede incluir elementos duplicados. Se utiliza normalmente cuando se necesita un control preciso de la posición de los elementos, y se usa la misma como un índice (entero) para acceder a los elementos. Esta interfaz es muy similar a la interfaz `Vector`, de las primeras versiones de java. En esta práctica trabajamos con una implementación concreta de `List` (`ArrayList`).
- `Queue`: una colección que se utiliza principalmente para almacenar elementos antes de su procesado. Además de las operaciones básicas sobre colecciones una cola proporciona operaciones adicionales de inserción, extracción e inspección. Habitualmente los elementos se ordenan tal y como llegan, de modo que se extraen en el mismo orden de entrada (FIFO, first-in, first-out), pero esta no es la única posibilidad, por ejemplo se pueden tener colas con prioridad, que ordenan elementos según un comparador proporcionado. Cualquiera que sea el criterio de ordenación se entiende que la cabecera de la cola es el elemento que se recuperará cuando se realice una solicitud de recuperación. Así en una cola FIFO cada nuevo elemento se inserta en la última posición, mientras que en otras se pueden insertar en diferentes posiciones. Cada implementación concreta de una cola debe especificar sus propiedades de ordenación.

NOTA: En telecomunicaciones estos conceptos son especialmente relevantes debido a que los algoritmos de gestión de colas de mensajes en los equipos de comunicación influyen notablemente en las características de los sistemas de comunicación.

- `Deque`: de forma similar a la anterior se utiliza principalmente para almacenar elementos antes de su procesado y añade operaciones a las básicas. Se pueden utilizar tanto con ordenación FIFO como en modo pila (LIFO, last-in, first-out), de modo que cada nuevo elemento se puede insertar, recuperar o eliminar de cualquiera de los dos extremos.
- `Map`: es un objeto que permitirá relacionar claves únicas con valores, por lo que las claves no pueden estar duplicadas y sólo pueden identificar un valor. Es muy similar a `Hashtable` de las primeras versiones de Java.
- `SortedMap`: es un tipo particular de mapa que mantiene sus elementos en orden ascendente según su clave. Usados cuando los pares clave/valor de la colección tienen esa naturaleza ordenada, como un diccionario o un listín telefónico.

En las últimas versiones de Java existen tres métodos para recorrer colecciones:

- 1) Usando operaciones de agregación (que queda fuera de los contenidos tratados en esta práctica)
- 2) Con bucles for-each
- 3) Usando Iteradores (**Iterators**)

En la versión 7, la utilizada en las prácticas, sólo se pueden utilizar las dos últimas. Previamente vio un ejemplo de uso de un iterador, el uso del bucle for-each lo puede ver en el ejemplo en:

```
System.out.println("Listado de la colección:\n\n");
for (Figura fig : serieDeFiguras) {
    System.out.println(fig);
}
```

5. EJERCICIOS

Debe haber descomprimido el fichero proporcionado y analizado detenidamente las clases proporcionadas, puede comprobar que la Figura 1 representa la estructura del código proporcionado.

Mantenga abierto el fichero “Colecciones” que contiene la clase principal para analizar qué está ocurriendo en el ejemplo.

A continuación ejecute make, que compilará y ejecutará el código ejemplo proporcionado. Si quiere disponer de la documentación en html también puede ejecutar make -f makeJavadoc .

Cuando la ejecución pare, esperando un return/intro, analice qué ha ocurrido y qué código se corresponde a ese comportamiento. Vuelva a repetir la operación cada vez que realice un ejercicio propuesto y haya modificado el código.

- Observe la declaración de la colección, `serieDeFiguras` será una referencia a una colección de tipo `ArrayList`, de elementos que serán implementaciones/realizaciones de la interfaz `Figura`. Observe también el uso del método `isEmpty` para verificar que al crearse está vacía.

```
/*
 * Este ejemplo utiliza una colección de tipo ArrayList,
 * que es una implementación de la interfaz List.
 */
List<Figura> serieDeFiguras = new ArrayList<Figura>();

*****
*****COMPROBACIÓN DE QUE ESTÁ VACÍA: isEmpty *****
*****
System.out.println("\n*****");
System.out.println( "Se ha creado la colección (ArrayList)");
System.out.println( "Se comprueba que esta vacía. ");
System.out.println("*****\n");
if(serieDeFiguras.isEmpty())
    System.out.println("NO hay elementos en la colección.\n");
else
    System.out.println("SI hay elementos en la colección.\n");
```

Código 4. Creación de un `ArrayList` de elementos de tipo `Figura` y uso del método `isEmpty` en el `main` de la aplicación (clase Colecciones).

- A continuación observe como crear figuras, añadirlas a la colección y verificar el tamaño. ¿Qué métodos se han utilizado? Piense ¿Qué

diferencia práctica cree que habrá entre realizar el new previamente y luego usar la referencia al objeto en el add, como se hace con cuad1, 2 y 3, o incluir el new en el método de añadir, como se hace con el resto de figuras? ¿Cuándo usaría una u otra? Observe como puede imprimirse SerieDeFiguras con el método println.

```
/*
*****AÑADIR ELEMENTOS A LA COLECCIÓN: add *****/
/*
 * Se crean dos objetos del tipo Cuadrado y se añaden a la lista.
 */
Figura cuad1 = new Cuadrado (3.5f,Color.ORANGE);
Figura cuad2 = new Cuadrado (2.2f,Color.YELLOW);
serieDeFiguras.add (cuad1);
serieDeFiguras.add (cuad2);

/*
***** COMPROBACIÓN DE QUE NO ESTÁ VACÍA: isEmpty *****
***** SE MUESTRA EL NÚMERO DE ELEMENTOS: size *****
*****/

System.out.println("\n*****");
System.out.println( "EJEMPLO: acabo de meter dos elementos," );
System.out.println( "Se comprueba que no esta vacia. " );
System.out.println("*****\n");

if(serieDeFiguras.isEmpty())
    System.out.println("NO hay elementos en la colección");
else
    System.out.println("SI hay elementos en la colección, concretamente "
        + serieDeFiguras.size()+"\n");
try{
    System.out.println("Pulse intro para continuar");
    System.in.read();
}catch(IOException exc){
    System.err.println(exc+"Error al leer\n\n");
}

/*
 * Se crea un nuevo cuadro del tipo OtroCuadrado y se añade a la lista.
 */
Figura cuad3 =      new OtroCuadrado (8.9f,"AMARILLO SERENIDAD");
serieDeFiguras.add (cuad3);

/*
 * Se crea un círculo del tipo Circulo y se añade a la lista.
 */
serieDeFiguras.add (new Circulo (3f, Color.BLACK));

/*
 * Se crea un círculo del tipo OtroCirculo y se añade a la lista.
 */
serieDeFiguras.add (new OtroCirculo (4f,"MELOCOTON MADURO"));

/*
 * Se crean dos rectángulos del tipo Rectangulo y se añaden a la lista.
 */
serieDeFiguras.add (new Rectangulo (2f, 3f, "CUARZO ROSA"));
serieDeFiguras.add (new Rectangulo (12f, 3f,"LILA GRIS"));
```

Código 5. Instanciación de distintos tipos de figuras e inserción en la lista.

EJERCICIO 1: INSTANCIE 3 FIGURAS MÁS; UNA DEL TIPO RECTÁNGULO DE COLOR NEGRO, UNA DEL TIPO OTROCÍRCULO DE COLOR NEGRO Y OTRA DEL TIPO OTROCUADRADO DE COLOR BLANCO. AÑÁDALAS A LA COLECCIÓN Y MUÉSTRELA POR PANTALLA

- A continuación observe como puede comprobar si un objeto pertenece a la colección y la primera posición en la que aparece. ¿En qué posición aparece cuad3?

```
System.out.println("\n*****");
System.out.println(" EJEMPLO: Se busca un objeto usando su referencia, *");
System.out.println("           y se muestra su posicion.      *");
System.out.println("*****\n");

if(serieDeFiguras.contains(cuad3))
    System.out.println(cuad3
                        + " SI pertenece a la colección y esta en la posición "
                        + serieDeFiguras.indexOf(cuad3)+"\n\n");
else
    System.out.println(cuad3 + " NO pertenece a la colección\n\n");
```

Código 6. Usar la referencia de un elemento para verificar si está y saber la posición de la primera aparición. Método principal de la clase **Colecciones**.

- Ahora observe como borrar un elemento de la colección.

```
System.out.println("\n*****");
System.out.println(" EJEMPLO: Se borra el elemento usando su referencia, *");
System.out.println("           y se vuelve a buscar.      *");
System.out.println("*****\n");

serieDeFiguras.remove(cuad3);
if(serieDeFiguras.contains(cuad3))
    System.out.println(cuad3
                        + " SI pertenece a la colección y esta en la posición "
                        + serieDeFiguras.indexOf(cuad3)+"\n\n");
else
    System.out.println(cuad3+" NO pertenece a la colección\n\n");
```

Código 7. Borrar elementos usando su referencia. Método principal de la aplicación, clase **Colecciones**.

EJERCICIO 2: VUELVA A AÑADIR CUAD3 Y REPITA LA OPERACIÓN DE BÚSQUEDA Y LOS MISMOS MENSAJES ANTERIORES. OBSERVE LA POSICIÓN DE LA FIGURA ANTES Y DESPUÉS ¿QUÉ HA OCURRIDO? ¿Quién ocupa ahora la posición 2? ¿Qué conclusiones obtiene?

- Observe que un **ArrayList** puede contener objetos duplicados y que existe un método para mostrar la última ocurrencia de un objeto.

```

System.out.println("\n*****");
System.out.println(" EJEMPLO: Elementos duplicados.          *");
System.out.println("           -Se vuelve a insertar cuad3,      *");
System.out.println("           el objeto esta dos veces      *");
System.out.println(" *****\n");

serieDeFiguras.add (cuad3);
if(serieDeFiguras.contains(cuad3)) {
    System.out.println(cuad3
        + " esta en la posicion "+serieDeFiguras.indexOf(cuad3)
        + " y tambien esta en la posicion "
        + serieDeFiguras.lastIndexOf(cuad3) + "\n\n");
} else{
    System.out.println(cuad3+" NO pertenece a la colección\n\n");
}

```

Código 8. Mostrar la última ocurrencia de un elemento usando su referencia. Método principal de la aplicación.

- Observe como recorrer la colección con el bucle for-each, imprimiendo por pantalla las figuras. ¿Con qué formato se muestra cada figura? Observe cómo se sobrescribió el método `toString` en la clase `Mifigura`.

```

System.out.println("\n*****");
System.out.println(" EJEMPLO: Uso de bucle for-each, para imprimir      *");
System.out.println("           las figuras por pantalla.          *");
System.out.println(" *****\n");

System.out.println("Listado de la colección:\n");
for (Figura fig : serieDeFiguras) {
    System.out.print("\t"+ fig);
}
System.out.println();

```

Código 9. Recorrer una lista con el bucle for-each. Método principal de la aplicación.

```

public String toString(){
    int indicepunto=this.getClass().getName().lastIndexOf(".");
    return this.getClass().getName().substring(indicepunto+1)
        + " de color "
        + this.getColor()
        + "\n";
}

```

Código 10. Sobreescritura del método `toString` en la clase `Mifigura`.

EJERCICIO 3: MUESTRE POR PANTALLA UNICAMENTE LOS COLORES DE LA COLECCIÓN, UTILIZANDO EL BUCLE FOR-EACH.

- Observe cómo puede hacer el cálculo del área total de la colección, usando el bucle for-each. Observe la característica de polimorfismo. Se invoca siempre igual al método `getArea` pero cada vez se ejecuta un código diferente, según el tipo concreto de la referencia `tmp`.

```

System.out.println("\n*****");
System.out.println(" EJEMPLO 2: Uso de bucle for-each, para calcular      *");
System.out.println("           el area total.                                *");
System.out.println(" *****\n");

float areaTotal = 0.f;

for (Figura tmp: serieDeFiguras) {
    areaTotal = areaTotal + tmp.getArea();
}

System.out.println ( "Tenemos un total de " + serieDeFiguras.size()
                    + " figuras y su area total es de "
                    + areaTotal + " uds cuadradas");

```

Código 11. Uso del bucle for each para calcular el área total. Método principal de la aplicación.

EJERCICIO 4: CACULE EL PERIMETRO TOTAL DE LA COLECCION UTILIZANDO UN BUCLE FOR EACH Y MUÉSTRELO POR PANTALLA

- A continuación observe cómo recorrer la colección con un iterador, mostrando cada figura por pantalla.

```

System.out.println("\n*****");
System.out.println(" EJEMPLO: Se obtiene un iterador y se muestran      *");
System.out.println("           los objetos.                                *");
System.out.println(" *****\n");

Iterator<Figura> iterador=serieDeFiguras.iterator();
while(iterador.hasNext()){
    System.out.print(iterador.next());
}

```

Código 12. Uso del iterador en el método principal de la aplicación.

EJERCICIO 5: MUESTRE POR PANTALLA EL ÁREA DE CADA FIGURA NEGRA DE LA COLECCIÓN USANDO UN ITERADOR Y CUÉNTELAS, MOSTRANDO POR PANTALLA LA CANTIDAD TOTAL DE FIGURAS NEGRAS.

- Ahora observe el uso de algoritmos genéricos (funciones estáticas de la clase **Collections**). En este caso el método usado invierte el orden de una colección. Fíjese bien ¿cómo se realiza la invocación al método **reverse**? Recuerde las características de los métodos estáticos o de clase.

```

System.out.println("\n*****");
System.out.println(" EJEMPLO: uso el algoritmo sort para dar la vuelta      *");
System.out.println("           a la colección.                                *");
System.out.println(" *****\n");

System.out.println("Orden original \n"+serieDeFiguras);
Collections.reverse(serieDeFiguras);
System.out.println("\nNuevo Orden \n"+serieDeFiguras);

```

Código 13. Uso de la función estática **reverse** de la clase **Collections** en el método principal de la aplicación.

- Observe la construcción (Código 3) y uso (Código 14) de un comparador “personalizado” para comparar figuras por su área y ordenarlas de mayor a menor área. Este diseño de la librería permite la creación de algoritmos personalizados a los requisitos del programador facilitando además su reutilización.

```

System.out.println("\n*****");
System.out.println(" EJEMPLO: Uso de comparador personalizado.      *");
System.out.println("           Realiza la ordenacion por area      *");
System.out.println("           de mayor a menor.                  *");
System.out.println("*****\n");

OrdenacionArea ordenador=new OrdenacionArea();
Collections.sort(serieDeFiguras,ordenador);
System.out.println("\nAhora ordenadas por Area, de mayor a menor \n");
for (Figura tmp: serieDeFiguras) {
    System.out.println(tmp+": Area= "+tmp.getArea());
}

```

Código 14. Uso del algoritmo de comparación implementado en `OrdenacionArea` en la función estática `sort` de la clase `Collections` (método principal de la aplicación).

EJERCICIO 6: CREE UN COMPARADOR PARA ORDENAR POR PERÍMETRO DE MENOR A MAYOR. UTILÍCELO PARA ORDENAR LA COLECCIÓN Y MUÉSTRELA POR PANTALLA, MOSTRANDO EL PERÍMETRO DE CADA FIGURA.

Trabajo a Entregar

Comprima la carpeta que contiene todo el código correspondiente a la práctica, que incluya los ejercicios realizados con el makefile correspondiente para realizar la compilación y ejecución, en un fichero .zip de nombre su uvus. Entregue el fichero en la tarea correspondiente.

6. REFERENCIAS

- [1] <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>
- [2] <https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html>
- [3] <https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>
- [4] <http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html>
- [5] <http://docs.oracle.com/javase/tutorial/collections/index.html>
- [6] <http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html>
- [7] <https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>
- [8] <https://docs.oracle.com/javase/7/docs/api/java.awt/Color.html>



PRACTICA 11:Map y HashMap.

1. OBJETIVO

El objetivo de esta práctica es entender la utilidad de los mapas de tipo `HashMap` en java y la forma de tratar con ellos para las operaciones básicas como son la creación, inserción y borrado de objetos, recuperación de valores, iteración, limpieza y obtención del tamaño.

2. MAPAS

Un mapa es una colección de datos de la forma “clave-valor”. En una tabla o matriz, el dato almacenado se referencia mediante un índice que indica la posición dentro de la tabla o la matriz. Por ejemplo, para referirnos al elemento 5 de una tabla se indica el índice dentro de la tabla. Como los índices comienzan en 0 el elemento 5 de la tabla `tab` sería `tab[4]`. Se podría decir que la clave en una tabla es el índice del elemento y que el valor es el propio elemento. En los mapas, para indicar un determinado elemento hacemos referencia a su clave, y conociendo la clave, podemos obtener el valor. Por ejemplo, si queremos almacenar una colección de libros de una biblioteca, la clave podría ser el ISBN del libro y el valor un objeto con todos los datos correspondientes al libro. Otro ejemplo podría ser una colección de usuarios, donde la clave puede ser el DNI y el valor un objeto con todos los datos necesarios del usuario.

La interfaz `Map` de java define operaciones que son soportadas por un conjunto de asociaciones clave-valor en el que las claves son únicas. `HashMap` es una implementación de la interfaz `Map`. Hay otras implementaciones de la interfaz `Map` que no se van a ver como son `TreeMap` y `LinkedHashMap` (la diferencia entre estas clases es la forma en que se guardan los valores en el mapa).

3. OPERACIONES

Las operaciones que se pueden realizar sobre un `HashMap` se presentan a continuación.

Creación de HashMap

Para la creación de un `HashMap` se especifica el tipo de la clave y el tipo del valor. Por ejemplo, en:

```
HashMap<String,Double> productos = new HashMap<String,Double>();
```

se está declarando la variable `productos` como un `HashMap` (una colección de `productos`), donde la clave será un objeto de tipo `String` que será el código del producto y el valor un objeto de tipo `Double` que será el precio del producto. Además se está creando el `HashMap`.

O en:

```
HashMap<Dni, Usuario> usuarios = new HashMap<Dni, Usuario>();
```

se está declarando la variable usuarios como un HashMap (una colección de usuarios), donde la clave será un objeto de tipo Dni y el valor un objeto del tipo Usuario. Además se está creando el HashMap.

Inserción de elementos en el HashMap

Para la inserción de elementos en un HashMap se utiliza el método put en el que se indica la clave y el valor del elemento a insertar. Por ejemplo:

```
productos.put("DISK", new Double(100.5));
```

añade un nuevo producto al HashMap productos con el código "DISK" y el precio 100.5.

Recuperación de valores del HashMap

La recuperación del valor de un elemento del HashMap se realiza con el método get indicando la clave del valor que se quiere recuperar. Por ejemplo:

```
Double c = productos.get("DISK");
```

recupera el precio del producto cuyo código es "DISK".

Iteración en el HashMap

Otra forma de recuperar los valores del HashMap es iterando sobre el HashMap completo. Para este propósito se utiliza un iterador (Iterator). Para obtener un iterador para las claves, primero es necesario obtener el conjunto de claves mediante el método KeySet del HashMap. Por ejemplo, con el siguiente código se muestran todos los productos del HashMap productos:

```
String clave;  
  
Iterator<String> iteradorProductos = productos.keySet().iterator();  
  
while (iteradorProductos.hasNext()) {  
  
    clave = iteradorProductos.next();  
  
    System.out.println(clave + " - " + productos.get(clave));  
  
}
```

Obtención del tamaño del HashMap

El número de elementos almacenados en el HashMap se puede saber mediante el método `size` del HashMap. Por ejemplo:

```
System.out.println("Productos en el HashMap: " + productos.size());
```

imprime el número de productos que hay almacenados en el HashMap `productos`.

Limpieza del HashMap

La limpieza del HashMap se realiza con el método `clear()` del HashMap. Por ejemplo:

```
productos.clear();
```

elimina todos los elementos del HashMap `productos`.

Comprobando si el HashMap contiene una clave o valor

Para comprobar si el HashMap contiene una determinada clave o un determinado valor, se utilizan los métodos `containsKey` y `containsValue` respectivamente. Por ejemplo:

```
String codigo = "DISPLAY";  
  
if (productos.containsKey(codigo)) {  
  
    System.out.println("El precio del producto es:" +  
  
                      productos.get(codigo));  
  
}  
  
else{  
  
    System.out.println("No hay ningún producto con ese código.");  
  
}
```

Indica si el producto cuyo código es “DISPLAY” se encuentra en el HashMap `productos`.

Comprobando si el HashMap está vacío

Para comprobar si el HashMap está vacío se utiliza el método `isEmpty`. Por ejemplo:

```
boolean vacio = productos.isEmpty();  
  
if(vacio){  
  
    System.out.println("No hay productos");  
  
} else {  
  
    System.out.println("Hay productos");  
  
}
```

Indica si hay elementos en el HashMap productos.

Borrando objetos del HashMap

Para borrar objetos del HashMap se usa el método `remove` proporcionando la clave. Por ejemplo:

```
if (productos.containsKey("DISK")) {  
  
    productos.remove("DISK");  
  
}  
  
else{  
  
    System.out.println("No hay ningun producto con ese codigo.");  
  
}
```

elimina el elemento del HashMap productos cuya clave es "DISK".

4. Ejemplo de HashMap

En el siguiente ejemplo se ha creado una clase `GestorAlmacen` que contiene el método `main` y es la encargada de gestionar el almacén, añadiendo los productos que introduce el usuario, modificando el precio, mostrando todos los productos o borrando un producto. Para la lectura de datos de la entrada estándar se usa un objeto de la clase `Scanner` que permite al usuario introducir la opción elegida y los datos necesarios para cada opción. La clase `Almacen` es la clase que contiene el `HashMap` con los productos del almacén.

A continuación se muestran tanto la clase GestorAlmacen como Almacen.

```
/*
 * Fichero: GestorAlmacen.java
 *
 * Fundamentos de Programacion II. GIT.
 * Departamento de Ingenieria Telematica
 * Universidad de Sevilla
 */
 */

package fp2.poo.practical1;

import java.util.Scanner;

/**
 * Descripción: Esta es una clase de ejemplo para presentar
 * los HashMap.
 *
 * @version version 1.0 Abril 2016
 * @author Fundamentos de Programacion II
 */
public class GestorAlmacen {

    /**
     * Este metodo es el método principal de la aplicación
     * Crea un objeto del tipo Almacen y realiza operaciones sobre él
     */
    public static void main( String args[] ) {

        Almacen almacenDeProductos = new Almacen();
        Scanner sc = new Scanner(System.in);
        int opcionElegida = 0;
        float precio;
        String codigo;

        while (opcionElegida != 5){
            System.out.println("Introduce el numero de la opcion que
quieras:");
            System.out.println("1.- Introducir producto");
            System.out.println("2.- Modificar precio");
            System.out.println("3.- Mostrar todos los productos");
            System.out.println("4.- Eliminar producto");
            System.out.println("5.- Salir");
            opcionElegida = sc.nextInt();

            switch (opcionElegida){
                case 1:
                    System.out.println("Introduce el codigo del producto:");
                    codigo = sc.next();
                    System.out.println("Introduce el precio del producto:");
                    precio = sc.nextFloat();
                    almacenDeProductos.guardaProducto(codigo, precio);
                    break;
                case 2:
                    System.out.println("Introduce el codigo del producto del
que quieres cambiar el precio:");
                    codigo = sc.next();
                    if (almacenDeProductos.existeProducto(codigo)){
                        System.out.println("Introduce el precio del
producto:");
                        precio = sc.nextFloat();
                    }
            }
        }
    }
}
```

```

                almacenDeProductos.modificaPrecio(codigo, precio);
            }
        else{
            System.out.println("No hay ningun producto con ese
codigo.");
        }

        break;
    case 3:
        almacenDeProductos.muestraProductos();
        break;
    case 4:
        System.out.println("Introduce el codigo del producto que
quieres eliminar:");
        codigo = sc.next();
        almacenDeProductos.eliminaProducto(codigo);
        break;
    case 5:
        break; // Si la opcion es 5 no se hace nada
    default:
        System.out.println("Tienes que introducir una opcion
valida");
    }
}

}

```

GestorAlmacen.java

```

/*
 * Fichero: Almacen.java
 *
 * Fundamentos de Programacion II. GITT.
 * Departamento de Ingenieria Telematica
 * Universidad de Sevilla
 */

package fp2.poo.practical1;

import java.util.HashMap;
import java.util.Iterator;

/**
 * Descripcion: Esta es una clase de ejemplo de HashMap.
 *              Un almacén contiene una colección de productos
 *              identificados por un código de producto
 *
 *
 * @version version 1.0 Abril 2016
 * @author Fundamentos de Programacion II
 */
public class Almacen {

    /** Atributo privado donde se almacenan los productos del almacén. */
    private HashMap<String,Float> productos;

    /**
     * Constructor de la clase Almacén.
     *
     * Parametros: No hay parámetros.
     */
    public Almacen() {

```

```

    // Crea el HashMap para la lista de productos
    this.productos = new HashMap<String,Float>();;
}

/**
 * Indica si un producto existe en el almacén
 */
public boolean existeProducto(String codigo){

    boolean resultado = false;
    if (productos.containsKey(codigo)){
        resultado = true;
    }

    return resultado;
}

/**
 * Guarda un producto en el almacén
 */
public void guardaProducto(String codigo, float precio){

    if (productos.containsKey(codigo)){
        System.out.println("No se puede introducir el producto. El
codigo esta repetido.");
    }
    else{
        productos.put(codigo, precio);
    }
}

/**
 * Modifica el precio de un producto del almacén
 */
public void modificaPrecio(String codigo, float precio){

    if (productos.containsKey(codigo)){
        productos.put(codigo, precio);
    }
    else{
        System.out.println("No hay ningun producto con ese codigo.");
    }
}

/**
 * Muestra los productos del almacén
 */
public void muestraProductos(){

    String clave;
    Iterator<String> iteradorProductos
        = productos.keySet().iterator();

    System.out.println("Hay los siguientes productos en el almacen:");
    while(iteradorProductos.hasNext()){
        clave = iteradorProductos.next();
        System.out.println(clave + " - " + productos.get(clave));
    }
}

/**
 * Elimina un producto del almacén
 */
public void eliminaProducto(String codigo){

```

```

        if (productos.containsKey(codigo)) {
            productos.remove(codigo);
        }
        else{
            System.out.println("No hay ningun producto con ese codigo.");
        }
    }

}

```

Almacen.java

Ejercicios.

1. Descargue el código proporcionado para esta práctica de la plataforma de enseñanza virtual y compile y ejecute con make para ver el funcionamiento del GestorAlmacen (los números decimales se introducen con coma). El Makefile proporcionado es el siguiente:

```

#
# Makefile ejemplo almacen
#
# Entorno.

DIRSRC = ./src/
DIRBIN = ./bin/
DIRJAR = ./jar/

# Clases.

RUTACLASES = fp2/poo/practical11/

CLASEALMACEN    = Almacen
CLASEMAIN      = GestorAlmacen

PRINCIPAL = fp2.poo.almacen.GestorAlmacen

ejecuta: $(DIRJAR)$(CLASEMAIN).jar
        java -jar $(DIRJAR)$(CLASEMAIN).jar

$(DIRJAR)$(CLASEMAIN).jar: $(DIRBIN)$(RUTACLASES)$(CLASEALMACEN).class \
                           $(DIRBIN)$(RUTACLASES)$(CLASEMAIN).class

        jar cvfe $(DIRJAR)$(CLASEMAIN).jar $(PRINCIPAL) \
              -C $(DIRBIN) $(RUTACLASES)$(CLASEMAIN).class \
              -C $(DIRBIN) $(RUTACLASES)$(CLASEALMACEN).class \
                           \
# MAIN
$(DIRBIN)$(RUTACLASES)$(CLASEMAIN).class:
$(DIRBIN)$(RUTACLASES)$(CLASEALMACEN).class\
                           \
$(DIRSRC)$(RUTACLASES)$(CLASEMAIN).java

        javac -g -Xlint -classpath $(DIRBIN) -encoding ISO-8859-1 -d
$(DIRBIN) -classpath $(DIRBIN) $(DIRSRC)$(RUTACLASES)$(CLASEMAIN).java

# ALMACEN
$(DIRBIN)$(RUTACLASES)$(CLASEALMACEN).class:
$(DIRSRC)$(RUTACLASES)$(CLASEALMACEN).java
        javac -g -Xlint -classpath $(DIRBIN) -encoding ISO-8859-1 -d
$(DIRBIN) -classpath $(DIRBIN) $(DIRSRC)$(RUTACLASES)$(CLASEALMACEN).java

```

Makefile

Incluya en el código las opciones correspondientes para “Mostrar el precio de un producto”, “Mostrar el numero de productos” y “Limpiar el almacén”. Para ello debe crear los métodos en la clase Almacen:

```
void mostrarPrecio(String codigo)  
int numProductos() y  
void limpiarAlmacen()
```

2. Basándose en el ejemplo anterior cree una clase GestorEmpresa en el paquete fp2.poo.empresa que permita dar de alta, bajar y modificar los datos de los empleados de la empresa entre otras operaciones. Para ello cree la clase Empresa que contenga un HashMap con los empleados de la empresa. La clave de los elementos del HashMap debe ser un String que representa el DNI del empleado y el valor debe ser un objeto del tipo Empleado que implementa la interfaz Persona ya dada en otra práctica. En el código dado para esta práctica ya se proporciona el fichero Persona.java, Empleado.java, y el Makefile necesario para la compilación y ejecución.

Trabajo a Entregar

Comprima el directorio que contiene el código de los ejercicios realizados y entréguelo en la plataforma de enseñanza virtual.

*Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla*

FUNDAMENTOS DE LA PROGRAMACIÓN II

Práctica Final de Programación Orientada a Objetos

@2023

Índice

1. Introducción

- 1.1 Objetivos.
- 1.2 Recomendaciones iniciales.

2. Presentación

- 2.1 Descripción del sistema.
- 2.2 Modelado del programa.

3. Especificación del Programa

- 3.1 Interfaces y excepciones utilizadas.
- 3.2 Argumentos de la Línea de Comandos.
- 3.3 Ficheros de entrada.
- 3.4 Formato del fichero de configuraciones.
- 3.5 Lectura de las URL bloqueadas del fichero de configuración.
- 3.6 Formato del fichero de solicitudes.
- 3.7 Lectura de las solicitudes del fichero de solicitudes.
- 3.8 Errores de ejecución.
- 3.9 Pruebas del programa.

4. Recomendaciones Finales

- 4.1 Organización del trabajo.
- 4.2 Normas para la defensa.
- 4.3 Documentación a entregar.
- 4.4 Fechas de entrega y de examen.

1. Introducción

1.1 Objetivos

Este documento contiene la definición de la práctica final de programación orientada a objetos que los alumnos de la asignatura de Fundamentos de la Programación II, que se presenten a alguna de las convocatorias del actual curso, deberán desarrollar y presentar en la fecha correspondiente según las normas de la asignatura.

El objetivo del trabajo es el aprendizaje del manejo de los conceptos básicos de la programación orientada a objetos, así como del entorno de trabajo utilizado. Para ello se propone la implementación de un programa, en el lenguaje de programación Java, que simule el funcionamiento de un sistema cuyo comportamiento se describe en este documento.

La información que procesará el programa, así como cualquier otra información auxiliar necesaria, se obtendrá mediante la lectura de ficheros o de la entrada estándar. Los resultados obtenidos después de ejecutar el programa se escribirán en la salida estándar (**System.out**), y en la salida de errores (**System.err**).

1.2. Recomendaciones iniciales

Esta práctica final utiliza los conceptos desarrollados durante las sesiones prácticas. Los ejemplos han sido orientados para facilitar la implementación de la práctica final de programación orientada a objetos.

Si algunos de los aspectos relativos al trabajo no le quedan claros, acuda al foro de la asignatura disponible en la plataforma de *enseñanza virtual* de la *Universidad de Sevilla*, donde se centralizarán todas las preguntas y respuestas relacionadas con el trabajo. Si en el foro no encuentra la respuesta, plantee en él la consulta.

También puede plantear las dudas a su profesor de prácticas, durante el desarrollo de las mismas o en horario de tutoría.

2. Presentación

2.1. Descripción del sistema.

El trabajo consistirá en modelar el funcionamiento de un **Proxy**. Un Proxy recibe **solicitudes** de recursos. Si el recurso no está almacenado localmente en el Proxy, lo obtiene de Internet si existe, en cuyo caso lo almacena localmente. Si el recurso ya está almacenado localmente en el Proxy, no es necesario buscarlo en Internet.

El programa debe manejar los siguientes objetos:

- **Proxy**: Un Proxy recibe solicitudes y las procesa.
- **Solicitud**: Una solicitud consta de la siguiente información
 - Número de entrada. Es un número secuencial que identifica la solicitud.
 - Protocolo. En este contexto se usará **https** o bien **http**.
 - Máquina. Una cadena que identifica a una máquina. Por ejemplo **www.us.es**.
 - Puerto. Un número entero que usualmente será **443** para **https** y **80** para **http**.
 - Directorio y fichero. Una cadena que identifica un fichero en un directorio. Por ejemplo, **/sites/default/files/2019-05/sello-US_0.jpg**.
- **URL**: Este objeto es un localizador de recurso uniforme (*Uniform Resource Locator*) y sirve para identificar un recurso en la web. En nuestro contexto, un recurso es un fichero.
- **Info**: Contiene la información con el número de accesos realizados a un recurso.
- **InfoRecursoLocal**: Contiene la información referente a un recurso local:
 - el número de accesos realizados y
 - el número de bytes.
- **RecursoLocal**: Contiene una URL y la información referente a un recurso local.
- **URLBloqueada**: URL a la que el Proxy no permite el acceso.
- **InfoBloqueada**: Contiene la información referente a una URL bloqueada.

2.2. Modelado del programa.

Se proporcionan interfaces y una clase abstracta que deberán ser implementadas por el alumno, para modelar el sistema descrito.

3. Especificación del Programa

El programa a realizar como práctica final de programación orientada a objetos de la asignatura de Fundamentos de la Programación II para la convocatoria del curso actual se ajustará a lo especificado en esta sección.

Todas las clases que el alumno implemente deben pertenecer al paquete **fp2.poo.pfpoox**, siendo **xxx** el uvus (usuario virtual de la Universidad de Sevilla) del alumno en la enseñanza virtual (ev). En el makefile proporcionado y en el código proporcionado aparece **fp2.poo.pfpoofp2**, y se deberán realizar las modificaciones oportunas.

3.1 Interfaces y excepciones utilizadas.

Se proporcionan las siguientes interfaces en el paquete **fp2.poo.utilidades**:

- **SolicitudInterfaz**
- **URLBloqueadaInterfaz**
- **RecursoLocalInterfaz**
- **InfoInterfaz**
- **InfoRecursoLocalInterfaz**
- **InfoBloqueadaInterfaz**

Se proporciona la siguiente clase abstracta en el paquete **fp2.poo.utilidades**:

- **ProxyAbstracta**

También se proporcionan las siguientes excepciones en el paquete **fp2.poo.utilidades.Excepciones**:

- **ErrorEnLosArgumentosDeEntradaExpcion**

Se lanzará esta excepción cuando el número de argumentos en línea de comandos sea distinto de dos. Estos argumentos indican el nombre del fichero de configuración y el nombre del fichero de solicitudes.

- **OperacionNoPermitidaExpcion**

Esta excepción se lanza cuando se intenta realizar una operación que no está permitida.

En esta práctica final se deben implementar todas las interfaces mencionadas anteriormente.

La clase abstracta **ProxyAbstracta** se implementará en la clase **Proxy**. La clase **Proxy** procesará un conjunto de **Solicitudes** agrupadas en una colección de objetos del tipo **SolicitudInterfaz**.

A continuación, se muestra el contenido de la clase abstracta **ProxyAbstracta**. Los métodos abstractos deben ser implementados.

```
/*
 *  @(#)ProxyAbstracta .java
 *
 *  Fundamentos de Programacion II. GITI.
 *  Departamento de Ingenieria Telematica
 *  Universidad de Sevilla
 *
 */
package fp2.poo.utilidades;

import fp2.poo.utilidades.Excepciones.OperacionNoPermitidaExcepcion;
import fp2.poo.utilidades.SolicitudInterfaz;

import java.net.MalformedURLException;
import java.net.URL;
import java.netURLConnection;

import java.io.File;
import java.io.IOException;
import java.io.BufferedInputStream;
import java.io.OutputStream;
import java.io.FileOutputStream;

/**
 * Esta es una clase que representa un Proxy.
 * Recibe solicitudes de recursos y las procesa.
 *
 * @version version 1.0 Abril 2023
 * @author Fundamentos de Programacion II
 */
public abstract class ProxyAbstracta {

    /**
     * Directorio en el que se encuentran los ficheros de la copia local.
     */
    public static final String COPIA_LOCAL      = "./copiaLocal/";

    /**
     * Este método realiza el procesamiento de las solicitudes de recursos
     * que recibe el Proxy.
     *
     * Para cada solicitud al Proxy, se comprueba si la solicitud corresponde
     * a una URL bloqueada, si es así, se incrementará un contador con el numero
     * de accesos realizados a esa URL bloqueada. En este caso, se muestra por
     * pantalla la cadena BLOCK seguida de la URL del recurso solicitado.
     *
     * Si el recurso solicitado no está bloqueado, se comprueba si está ya
     * almacenado en el Proxy. Si el recurso está almacenado en el Proxy,
     * se muestra por pantalla la cadena PROXY seguida de la URL del recurso solicitado.
     *
     * Si el recurso solicitado al Proxy no estuviera bloqueado ni almacenado
     * en el Proxy se busca en la web, y si existe entonces se obtiene y se
     * almacena en el directorio del Proxy copiaLocal. En este caso, se muestra
     * por pantalla la cadena OK seguida de la URL del recurso solicitado.
     *
     * Si el recurso solicitado al Proxy no estuviera bloqueado, tampoco estuviera
     * almacenado en la copia local, y no existe en la web, entonces se muestra
     * por pantalla la cadena NO_OK seguido de la URL del recurso solicitado.
     */
    public abstract void procesaSolicitudesDelCliente()
                      throws OperacionNoPermitidaExcepcion;

    /*
     * Este metodo muestra todas las solicitudes leidas del fichero de solicitudes.
     * La salida proporciona una linea por cada solicitud, y cada solicitud
     * debe mostrar el mismo formato y orden que el proporcionado en el fichero
     * de solicitudes que se proporciona al Proxy.
     */
}
```

```

/*
public abstract void muestraSolicitudes();

/*
 * Este metodo muestra todas las URL bloqueadas por el Proxy, leidas
 * del fichero de configuracion. Se genera una linea en la salida est ndar
 * por cada URL bloqueada por el Proxy, indicando el numero de veces que
 * ha sido solicitada seguido de la URL correspondiente.
 */
public abstract void muestraURLBloqueadas();

/*
 * Este metodo muestra todos los recursos almacenados localmente por el Proxy.
 * Genera una linea en la salida por cada recurso almacenado localmente,
 * indicando en primer lugar el n mero de veces que ha sido solicitada la URL
 * seguido por el tama o en bytes del fichero y la URL correspondiente.
 */
public abstract void muestraRecursos();

/*
 * Este metodo ordena los recursos almacenados en la copia local del Proxy
 * segun el numero de solicitudes recibidas de menor a mayor.
 * Para realizar esta ordenacion se debe usar la clase
 * OrdenacionRecursosPorAccesos de tal forma que implemente la interfaz
 * Comparator de Java.
 */
public abstract void ordenarRecursosPorAccesos();

/*
 * Este metodo obtiene un recurso de la web dada como parametro la URL
 * del recuso y lo guarda en un fichero local al proxy.
 *
 * Devuelve: el numero de bytes del fichero si existe, y -1 en caso de error.
 */
protected int guardarRecursoEnLocal(URL url){
    int contadorDeBytes = 0;
    try{
        String file = url.getFile();
        int pos = file.lastIndexOf("/");
        String camino = file.substring(pos + 1, file.length());

        URLConnection yc = url.openConnection();
        BufferedInputStream in = new BufferedInputStream(yc.getInputStream());
        int valor;
        OutputStream os = null;
        os = new FileOutputStream(ProxyAbstracta.COPIA_LOCAL + camino);
        while ((valor = in.read()) != -1) {
            contadorDeBytes++;
            os.write(valor);
        }
        in.close();
        os.close();
    }catch(MalformedURLException e){
        return -1;
    }catch(IOException e){
        return -1;
    }
    return contadorDeBytes;
}
}

```

ProxyAbstracta.java

La siguiente tabla presenta un resumen de las clases e interfaces y clases abstractas utilizadas, con los ficheros en los que se implementan.

<i>Clase</i>	<i>Fichero en que se implementa la clase</i>	<i>Interfaz/clase abstracta</i>	<i>Fichero en que se especifica la interfaz</i>
<code>Info</code>	<code>Info.java</code>	<code>InfoInterfaz</code>	<code>InfoInterfaz.java</code>
<code>InfoBloqueada</code>	<code>InfoBloqueada.java</code>	<code>InfoBloqueadaInterfaz</code>	<code>InfoBloqueadaInterfaz.java</code>
<code>InfoRecursoLocal</code>	<code>InfoRecursoLocal.java</code>	<code>InfoRecursoLocalInterfaz</code>	<code>InfoRecursoLocalInterfaz.java</code>
<code>URLBloqueada</code>	<code>URLBloqueada.java</code>	<code>URLBloqueadaInterfaz</code>	<code>URLBloqueadaInterfaz.java</code>
<code>RecursoLocal</code>	<code>RecursoLocal.java</code>	<code>RecursoLocalInterfaz</code>	<code>RecursoLocalInterfaz.java</code>
<code>Proxy</code>	<code>Proxy.java</code>	<code>ProxyAbstracta</code>	<code>ProxyAbstracta.java</code>

La clase **OrdenacionRecursosPorAccesos**, implementa la interfaz **Comparator** de Java y permite realizar la ordenación de los recursos según el número de accesos de menor a mayor.

La interfaz **Iterator** está implementada en la clase **LecturaConfiguracion**. **LecturaConfiguracion** permite la lectura de las URL bloqueadas por el Proxy desde un fichero (existen más comentarios en la sección Lectura de los Ficheros de Entrada).

La interfaz **Iterator** está implementada en la clase **LecturaSolicitudesDeEntrada**. **LecturaSolicitudesDeEntrada** permite la lectura de los recursos solicitados al Proxy y son leídas desde un fichero (existen más comentarios en la sección Lectura de los Ficheros de Entrada).

3.2 Argumentos de la Línea de Comandos.

El programa aceptará dos argumentos en línea de comandos. Estos argumentos son el nombre del fichero de configuración que contiene los datos de las URL bloqueadas por el Proxy, y el fichero de solicitudes que contiene las solicitudes de recursos realizadas al Proxy.

Si no se proporcionan los dos nombres de fichero se lanza la excepción **ErrorEnLosArgumentosDeEntradaExpcion**.

En caso de que el fichero de configuración o bien el fichero de solicitudes no exista, se lanza la excepción **OperacionNoPermitidaExpcion**.

En caso de que se lance una de estas dos excepciones se imprime por la salida de errores un mensaje y finaliza el programa. Los mensajes correspondientes son los siguientes:

- "Número de argumentos incorrecto.": Este mensaje se genera si el número de argumentos en la línea de comandos es distinto de 2.
- "Fichero inexistente.": Este mensaje se genera si el fichero no existe.

3.3 Ficheros de entrada.

Los ficheros de entrada cuyos nombres se proporcionan como argumentos en la línea de comandos se ubicarán en el directorio **./datos**. Y serán dos: uno de configuración que proporciona las URL bloqueadas, y otro de solicitudes que proporciona las solicitudes que llegan al Proxy.

3.4 Formato del fichero de configuraciones.

En el fichero de configuraciones se especifican los datos de cada URL bloqueada en una cadena de texto. Esta cadena contiene en el siguiente orden:

- Protocolo, que podrá ser **https** o bien **http**.
- Máquina, que identifica a una máquina.
- Puerto, que podrá ser **443** para el protocolo **https** y **80** para el protocolo **http**.
- Directorio y fichero, que identifica al fichero en el directorio indicado.

Un ejemplo de entrada del fichero de configuraciones es el siguiente:

http://trajano.us.es:80/graficos/esi.gif

en este ejemplo, se usa el protocolo **http**, con la máquina **trajano.us.es**, con el puerto **80**, del directorio **graficos**, y el fichero **esi.gif**.

En el fichero de configuraciones, cada URL bloqueada va en una única línea. Todas las entradas del fichero de configuración serán correctas.

3.5 Lectura de las URL bloqueadas del fichero de configuración.

Para la lectura del fichero de URL bloqueadas se proporciona la clase **LecturaConfiguracion**, (ya implementada) que implementa la interfaz **Iterator** que contiene dos métodos:

- **hasNext()**
(public boolean hasNext()) este método devuelve **true** si hay más URL bloqueadas en el fichero.
- **next()**
(public E next()) este método devuelve la siguiente URL en una referencia del tipo **URLBloqueadaInterfaz**, si no hay más entradas y se intenta obtener con este método se lanzará la excepción **NoSuchElementException** (que pertenece al paquete **java.util**).

De esta forma todas las URL bloqueadas se pueden obtener en el constructor de la clase **Proxy** de la siguiente forma (que no se proporciona en los ficheros entregados para la realización de la práctica final):

```
public class Proxy extends ProxyAbstracta{  
  
    private List<URLBloqueadaInterfaz> urlbloqueadas  
        = new ArrayList<URLBloqueadaInterfaz>();  
  
    private List<SolicitudInterfaz>      solicitudes  
        = new ArrayList<SolicitudInterfaz>();  
  
    private List<RecursoLocalInterfaz>    copiaLocal  
        = new ArrayList<RecursoLocalInterfaz>();  
  
    public Proxy(String nombreFicheroConf, String nombreFicheroSolicitudes)  
        throws OperacionNoPermitidaExpcion{  
  
        LecturaConfiguracion lecturaConf  
            = new LecturaConfiguracion(nombreFicheroConf);  
        while (lecturaConf.hasNext()) {  
            URLBloqueadaInterfaz urlBloquedas = lecturaConf.next();  
            urlbloqueadas.add(urlBloquedas);  
        }  
    }  
//la clase Proxy continúa ...
```

Extracto de la clase `Proxy.java`

3.6 Formato del fichero de solicitudes.

En el fichero de solicitudes se especifican los datos de cada URL solicitada en varias cadenas de texto. Cada solicitud está escrita en una línea y llevan el siguiente orden en la misma línea:

- Número de entrada, que es un número de secuencia que identifica la solicitud.
- Protocolo, que puede ser la cadena **https** o bien **http**.
- Máquina, que es una cadena que identifica a la máquina que proporciona el fichero.
- Puerto, que podrá ser **443** para el protocolo **https** y **80** para el protocolo **http**.
- Directorio y fichero, que indica la ruta en la máquina donde está el fichero, y el fichero correspondiente.

Un ejemplo de entrada del fichero de solicitudes es el siguiente:

1 **https www.us.es 443 /sites/default/files/2019-05/marca-US-principal.jpg**

en este ejemplo, la solicitud corresponde con el número de entrada **1**, se usa el protocolo **https**, con la máquina **www.us.es**, con el puerto **443**, del directorio **/sites/default/files/2019-05/**, y el fichero **marca-US-principal.jpg**.

Todas las entradas del fichero de entradas serán correctas.

3.7 Lectura de las solicitudes del fichero de solicitudes.

Para la lectura del fichero de solicitudes se proporciona la clase **LecturaSolicitudesDeEntrada**, (ya implementada) que implementa la interfaz **Iterator** que contiene dos métodos:

- **hasNext()**
(public boolean hasNext()) este método devuelve **true** si hay más solicitudes en el fichero.
- **next()**
(public E next()) este método devuelve la siguiente solicitud en una referencia del tipo **SolicitudInterfaz**, si no hay más entradas y se intenta obtener con este método se lanzará la excepción **NoSuchElementException** (que pertenece al paquete **java.util**).

De esta forma todas las solicitudes al Proxy se pueden obtener en el constructor de la clase **Proxy** de la siguiente forma (que no se proporciona en los ficheros entregados para la realización de la práctica final):

```
public class Proxy extends ProxyAbstracta{  
    private List<URLBloqueadaInterfaz> urlbloqueadas  
        = new ArrayList<URLBloqueadaInterfaz>();  
    private List<SolicitudInterfaz>      solicitudes  
        = new ArrayList<SolicitudInterfaz>      ();  
    private List<RecursoLocalInterfaz>    copiaLocal  
        = new ArrayList<RecursoLocalInterfaz>      ();  
    public Proxy(String nombreFicheroConf, String nombreFicheroSolicitudes)  
        throws OperacionNoPermitidaExpcion{  
        LecturaSolicitudesDeEntrada lectura  
            = new LecturaSolicitudesDeEntrada( nombreFicheroSolicitudes );  
        while (lectura.hasNext()){  
            SolicitudInterfaz solicitud = lectura.next();  
            solicitudes.add ( solicitud );  
        }  
    }  
//la clase Proxy continúa ...
```

Extracto de la clase `Proxy.java`

3.8 Resultados del programa.

Los resultados se proporcionarán por la salida estándar (`System.out`) mediante el método `println`. El método de la clase `Proxy, procesaSolicitudesDelCliente` (`public void procesaSolicitudesDelCliente() throws OperacionNoPermitidaExpcion`) procesa las solicitudes de recursos proporcionadas al Proxy.

Las solicitudes de recursos están almacenadas en el siguiente atributo de la clase Proxy:

```
private List<SolicitudInterfaz> solicitudes = new ArrayList<SolicitudInterfaz> ();
```

Para cada solicitud proporcionada al Proxy, se comprueba si la solicitud corresponde a una URL bloqueada, si es así, se incrementará un contador con el número de accesos realizados a esa URL bloqueada.

En el Proxy, las URLs bloqueadas se almacenan en el siguiente atributo del Proxy:

```
private List<URLBloqueadaInterfaz> urlbloqueadas = new ArrayList<URLBloqueadaInterfaz>();
```

Si el recurso solicitado no está bloqueado, se comprueba si está ya almacenado en el Proxy. La información relacionada con los recursos locales se almacena en el siguiente atributo del Proxy:

```
private List<RecursoLocalInterfaz> copiaLocal = new ArrayList<RecursoLocalInterfaz> ();
```

A continuación, se especifica el comportamiento del Proxy en los distintos casos:

- Si el recurso solicitado al Proxy estuviera bloqueado, se muestra por pantalla la cadena BLOCK, seguido del recurso solicitado. Por ejemplo,

```
BLOCK https://www.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg
```

Para indicar que el recurso indicado (<https://www.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg>) está bloqueado por el Proxy.

- Si el recurso solicitado al Proxy no estuviera bloqueado y estuviera almacenado en la copiaLocal, se muestra por pantalla la cadena PROXY, seguida de la URL. Por ejemplo,

```
PROXY https://www.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg
```

Para indicar que el recurso solicitado (<https://www.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg>) se ha obtenido de la copia local al Proxy.

- Si el recurso solicitado al Proxy no estuviera bloqueado, y no estuviera almacenado en la copiaLocal, se busca en la web y si existe en la web, entonces se obtiene y se almacena en el directorio del Proxy `copiaLocal`, y se guarda la información correspondiente a este recurso local en el `ArrayList copiaLocal`. En este caso se muestra por pantalla la cadena "OK" seguido de la URL. Por ejemplo,

```
_OK__ https://www.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg
```

Para indicar que el recurso (<https://www.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg>) se ha obtenido de la web y se ha guardado en la copia local.

- Si el recurso solicitado al Proxy no estuviera bloqueado, tampoco estuviera almacenado en la copia local, y **no** existe en la web, entonces se muestra por pantalla la cadena "NO_OK" seguido de la URL. Por ejemplo,

```
NO_OK https://xxx.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg
```

Para indicar que el recurso (<https://xxx.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg>) no se ha podido obtener de la web.

Los métodos del Proxy que muestran la información almacenada en el Proxy relacionada con las solicitudes, las URL bloqueadas y los recursos almacenados en la copia local, se especifican a continuación.

El método **muestraSolicitudes** (**public abstract void muestraSolicitudes()**) especificado en **ProxyAbstracta**, genera la salida de todas las solicitudes que se proporcionan al Proxy. La salida debe proporcionar una línea por cada solicitud, y de cada solicitud se debe mostrar el mismo formato y orden que el proporcionado en el fichero de solicitudes que se proporciona al Proxy. Un ejemplo de una línea de salida del método **muestraSolicitudes** se muestra a continuación:

```
1 https www.us.es 443 /sites/default/files/2019-05/sello-US_0.jpg
```

El método **muestraURLBloqueadas** (**public abstract void muestraURLBloqueadas()**) especificado en **ProxyAbstracta**, genera una línea en la salida estándar por cada URL bloqueada por el Proxy, indicando el número de veces que ha sido solicitada seguido de la URL correspondiente. Un ejemplo de una línea de salida del método **muestraURLBloqueadas** se muestra a continuación:

```
3 https://www.us.es:443/sites/default/files/2019-05/sello-US_0.jpg
```

El método **muestraRecursos** (**public abstract void muestraRecursos()**) especificado en **ProxyAbstracta**, genera una línea en la salida por cada recurso almacenado localmente, indicando en primer lugar el número de veces que ha sido solicitada la URL seguido por el tamaño en bytes del fichero y la URL correspondiente. Un ejemplo de una línea de salida del método **muestraRecursos** se muestra a continuación:

```
2 15624 https://www.us.es:443/sites/default/files/2019-05/marca-US-principal.jpg
```

3.9 Pruebas del programa.

El programa se probará mediante clases que contengan el método **main**. Estas clases estarán en el paquete **fp2.poo.principal**. Se proporciona una clase de ejemplo, que crea un Proxy y realiza unas operaciones en el Proxy.

Se proporciona un **makefile** que puede servir de ejemplo para la compilación. El alumno puede utilizar partes de este makefile conforme vaya realizando el desarrollo de su programa.

4 Recomendaciones finales

4.1 Organización del trabajo.

Los ficheros deben de estar organizados según el enunciado de la práctica.

4.2 Normas para la defensa.

Para evaluar la Práctica Final de Programación Orientada a Objetos el alumno deberá hacer y superar un examen del trabajo realizado. El alumno debe saber modificar su programa según nuevas especificaciones. Durante la defensa se solicitará al alumno que realice modificaciones sobre el código realizado.

4.3 Documentación a entregar.

El trabajo se entregará en la plataforma virtual según las normas de la asignatura. La compilación debe realizarse con un fichero makefile.

Se valorará positivamente la inclusión en el código de los comentarios javadoc y la entrega de la documentación generada mediante la herramienta javadoc en el directorio ./doc.

También se valorará positivamente la generación del fichero .jar.

4.4 Fechas de entrega y de examen.

Primera opción para superar la práctica final:

Examen: lunes 22 de mayo de 2023 (12:30-14:30)

Fecha de entrega para los alumnos que se presenten 22 de mayo de 2023: antes del 19 de mayo de 2023 a las 23:59.

Examen de 1ª convocatoria:

Examen: 31 de mayo de 2023 (11:00 a 13:00)

Fecha de entrega para los alumnos que se presenten el 31 de mayo de 2023: antes del 29 de mayo de 2023 a las 23:59.

Examen de 2ª convocatoria:

Examen: 12 de julio de 2023.

Fecha de entrega para los alumnos que se presenten el 12 de julio de 2023: antes del 10 de julio de 2023 a las 23:59.