Tabla de contenidos

Lista de tablas

Lista de figuras

Lista de ejemplos

Capítulo 1. Introducción a Java.

En este primer capítulo conocerás la historia de Java y cuales son sus principales característica. Además conocerás cuales son las herramientas de desarrollo para programar en Java y escribirás y compilarás tu primer programa en Java. El capítulo se cierra con unos consejos de estilo sobre la codificación en Java.

1.1. Historia. El perfil de Java.

Los padres de Java son James Gosling (emacs) y Bill Joy (Sun). Java desciende de un lenguaje llamado Oak cuyo propósito era la creación de software para la televisión interactiva. Las características de Oak eran:

- Pequeño.
- Robusto.
- Independiente de la máquina.
- Orientado a objetos.

El proyecto de televisión interactiva fracasó y el interés de los creadores de Oak se dirigió a Internet bajo el lema «La red es la computadora».

Los criterios de diseño de Java fueron:

- Independiente de la máquina.
- Seguro para trabajar en red.
- Potente para sustituir código nativo.

1.2. Características del lenguaje.

La principal característica de Java es la de ser un lenguaje compilado e interpretado. Todo programa en Java ha de compilarse y el código que se genera *bytecodes* es interpretado por una máquina virtual. De este modo se consigue la independencia de la máquina, el código compilado se ejecuta en máquinas virtuales que si son dependientes de la plataforma.

Java es un lenguaje orientado a objetos de propósito general. Aunque Java comenzará a ser conocido como un lenguaje de programación de applets que se ejecutan en el entorno de un navegador web, se puede utilizar para construir cualquier tipo de proyecto.

Su sintaxis es muy parecida a la de C y C++ pero hasta ahí llega el parecido. Java no es una evolución ni de C++ ni un C++ mejorado.

En el diseño de Java se prestó especial atención a la seguridad. Existen varios niveles de seguridad en Java, desde el ámbito del programador, hasta el ámbito de la ejecución en la *máquina virtual*.

Con respecto al programador, Java realiza comprobación estricta de tipos durante la compilación, evitando con ello problemas tales como el desbordamiento de la pila. Pero, es durante la ejecución donde se encuentra el método adecuado según el tipo de la clase receptora del mensaje; aunque siempre es posible forzar un enlace estático declarando un método como `final`.

Todas las instancias de una clase se crean con el operador `new()`, de manera que un *recolector de basura* se encarga de liberar la memoria ocupada por los objetos que ya no están referenciados. La *máquina virtual* de Java gestiona la memoria dinámicamente.

Una fuente común de errores en programación proviene del uso de punteros. En Java se han eliminado los punteros, el acceso a las instancias de clase se hace a través de *referencias*.

Además, el programador siempre está obligado a tratar las posibles excepciones que se produzcan en tiempo de ejecución. Java define procedimientos para tratar estos errores.

Java también posee mecanismos para garantizar la seguridad durante la ejecución comprobando, antes de ejecutar código, que este no viola ninguna restricción de seguridad del sistema donde se va a ejecutar.

También cuenta con un cargador de clases, de modo que todas las clases cargadas a través de la red tienen su propio espacio de nombres para no interferir con las clases locales.

Otra característica de Java es que está preparado para la programación concurrente sin necesidad de utilizar ningún tipo de biblioteca.

Finalmente, Java posee un *gestor de seguridad* con el que poder restringir el acceso a los recursos del sistema.

A menudo se argumenta que Java es un lenguaje lento porque debe interpretar los bytecodes a código nativo antes de poder ejecutar un método, pero gracias a la tecnología JIT, este proceso se lleva a cabo una única vez, después el código en código nativo se almacena de tal modo que está disponible para la siguiente vez que se llame.

1.3. Herramientas de desarrollo.

Las herramientas de desarrollo de Java se conocen como *Java Development Kit*(JDK). En el momento de escribir este trabajo las herramientas de desarrollo van por la versión 1.5. Estas herramientas se pueden descargar gratuitamente de http://java.sun.com.

Este conjunto de herramientas cuenta entre otros con un compilador de línea de comandos *javac*; la máquina virtual de Java con la que poder ejecutar aplicaciones *java*; una herramienta de documentación *javadoc*; y una herramienta para empaquetar proyectos *jar*. La utilidad de estas herramientas la iremos viendo con detalle en las siguientes secciones.

Un detalle importante a la hora de ejecutar aplicaciones Java es indicar a la *máquina virtual* el lugar donde debe buscar las clases que no forman parte del paquete básico. Esta dirección se le indica con la variable de entorno `CLASSPATH`. Por ejemplo, si estamos en un entorno `linux` deberemos indicar esta dirección con la siguiente instrucción en el `.bashrc`:

export CLASSPATH=/home/usuario/MisClasses

Aunque también se puede especificar en el momento de la ejecución indicándolo en el parámetro -cp de java en la línea de comandos.

Debes pensar en Java no solamente como un lenguaje de programación si no como un conjunto de tecnologías basadas en el mismo lenguaje. Este conjunto de tecnologías te permite escribir aplicaciones para gráficos, multimedia, la web, programación distribuída, bases de datos y un largo etcétera.

1.4. El típico primer programa

En el listado del Ejemplo 1-1 se muestra el típico primer programa. Cópialo a un fichero que lleve por nombre `HolaJava.java`.

Ejemplo 1-1. Primer programa en Java

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

Para compilar el programa escribe en la línea de instrucciones:

```
[belfern@anubis Java]$javac HolaJava.java
```

Si todo va bien no tendrá ningn mensaje de error. Para ejecutar el programa escribe en la línea de instrucciones:

```
[belfern@anubis Java]$java HolaJava
```

El resultado que se mostrará en la consola es:

```
Hola Java
```

Una de las primeras cosas que hay que tener en cuenta es que en Java se distingue entre mayúsculas y minúsculas. La primera línea es la declaración de una clase pública llamada `HolaJava`. Sólo puede haber una clase pública en un fichero con extensión `java`. Esta clase contiene un único metodo `main` que es también público, no devuelve ningún valor y recibe un array de tipo base `String`. La única instrucción con la que cuenta `main` es `System.out.println` que sirve para mostrar mensajes de texto por pantalla.

La clase que contenga el método `main` es la clase principal de la aplicación, y es en ese método donde se inicia la ejecución de la aplicación.

1.5. Recomendaciones

En Java existen ciertas reglas de codificación que son comúnmente utilizadas por los programadores. Conviene conocer y seguir estas reglas.

- Los nombre de las clases deben empezar por mayúscula.
- Los atributos y méodos de las clases deben empezar por minúsculas y si están formadas por varias palabras, se escriben sin espacios y la primera letra de cada palabra en mayúscula.
- Las instancias de las clases siguen la misma recomendación que los métodos y atributos.
- Las constantes se escriben en mayúsculas.

Capítulo 2. Sintaxis de Java.

En este capítulo se mostrarán las construcciones básicas del lenguajes Java. Comienza el capítulo mostrando los tres tipos de comentarios que proporciona Java. Los comentarios de documentación resultan de especial importancia a la hora de comentar código. Se muestra como utilizar la herramienta de documentación `javadoc`. El capítulo sigue con la sintaxis válida para los identificadores, y se listan las palabras reservadas del lenguaje. El capítulo se cierra con la enumeración de los separadores y sus usos.

2.1. Comentarios

En Java existen tres tipos de comentarios:

- Comentarios de una solo línea como en C++

```
// Esta es una línea comentada.
```

- Comentarios de bloques como en C.

```
/* Aquí empieza el bloque comentado

y aquí acaba */
```

- Comentarios de documentación.

```
/** Los comentarios de documentación se comentan de este modo */
```

2.2. Documentación.

Una tarea importante en la generación de código es su documentación. El código no debe únicamente ejecutarse sin errores si no que además debe estar bien documentado. Java facilita esta tarea utilizando ciertas etiquetas en los comentarios de documentación.

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

- `@author [Nombre y Apellidos del autor]`
- `@version [Información de la versión]`

- `@param [nombreDelParametro] [Descripción]`
- `@return [Descripción del parámetro devuelto]`
- `@exception [Excepción lanzada]`
- `@see [Referencia cruzada]`
- `@deprecated [Comentario de porque es obsoleto]`

Además, en los comentarios se puede insertar código html para resaltar la documentación.

Los comentarios de autor y versión se aplican sólo a las clases. Los comentarios de parámetros, retorno y excepciones se aplican sólo a los métodos. Los comentarios de referencias cruzadas y obsolescencias se pueden aplicar a clases, métodos y atributos. En Ejemplo 2-1 se muestra una clase con los comentarios de documentación.

Ejemplo 2-1. Un ejemplo con comentarios de documentación

```
/**
 *  Esta clase define un punto en un espacio de dos dimensiones.
 *  @author Óscar Belmonte Fernández
 *  @version 1.0, 27 de Octubre de 2004
 */
public class Punto {
  protected float x; /** @param x Coordenada x del punto */
  protected float y;

  /**
   *  Constructor por defecto
   */
  public Punto() {
    x = 0.0f;
    y = 0.0f;
  }

  /**
   *  Constructor con argumentos.
   *  @param x La coordenada 'x' del punto.
   *  @param y La coordenada 'y' del punto.
   */
  public Punto(float x, float y) {
    this.x = x;
    this.y = y;
  }

  /**
   *  Con esta función se recupera el valor de la coordenada solicitada
   *  @param coordenada La coordenada que se solicita 'x' o 'y'
   *  @return El valor de la coordenada
   *  @deprecated Esta función se eliminará en próximas versiones
```

```
 *  @see #getX()
 *  @see #getY()
 */
public float get (String coordenada) {
  if(coordenada.equals("x")) return x;
  else if(coordenada.equals("y")) return y;
}

/**
 *  Esta función devuelve el valor de la coordenada 'x'
 *  @return El valor de la coordenada 'x'
 */
public float getX() {
  return x;
}

/**
 *  Esta función cambia el valor de la coordenada 'x'
 *  @param x El nuevo valor de la coordenada 'x'
 */
public void setX(float x) {
  this.x = x;
}

/**
 *  Esta función cambia el valor de la coordenada 'y'
 *  @param x El nuevo valor de la coordenada 'y'
 */
public void setY(float y) {
  this.y = y;
}

/**
 *  Esta función devuelve el valor de la coordenada 'y'
 *  @return El valor de la coordenada 'y'
 */
public float getY() {
  return y;
}

/**
 *  Esta función devuelve un punto que es el inverso del punto
 *  @return Punto El inverso de este punto
 */
public Punto inverso() {
  return new Punto(-x, -y);
}
/**
 *  Sobre escritura de toString()
 *  @return String Una cadena con la información a mostrar
 */
public String toString() {
  return "("+x+", "+y+")";
```

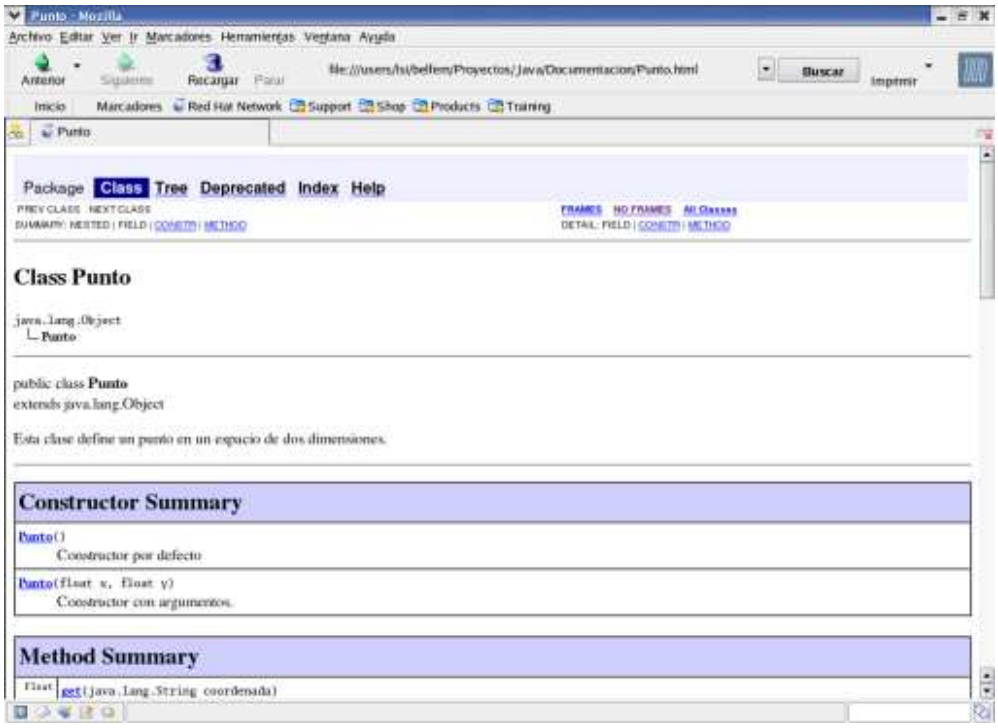
```
  }
}
```

Para ver como se genera la documentación copia el código del Ejemplo 2-1 en un fichero llamado `Punto.java` y crea un directorio llamado `Documentacion` en el directorio donde se encuentre ese fichero. En la línea de instrucciones escribe:

```
[belfern@anubis Java]$javadoc -d Documentacion Punto.java
Loading source file Punto.java...
Constructing Javadoc information...
Standard Doclet version 1.4.2_02
Generating Documentacion/constant-values.html...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating Documentacion/overview-tree.html...
Generating Documentacion/index-all.html...
Generating Documentacion/deprecated-list.html...
Building index for all classes...
Generating Documentacion/allclasses-frame.html...
Generating Documentacion/allclasses-noframe.html...
Generating Documentacion/index.html...
Generating Documentacion/packages.html...
Generating Documentacion/Punto.html...
Generating Documentacion/package-list...
Generating Documentacion/help-doc.html...
Generating Documentacion/stylesheet.css...
```

Entre otros se te habrá generado el fichero Punto.html, échale un vistazo con un navegador Web, verás algo parecido a lo que se muestra en la siguiente imagen:

Figura 2-1. Documentación vista en un navegador.



2.3. Identificadores

En Java los identificadores comienzan por una letra del alfabeta inglés, un subrayado «_» o el símbolo de dolar «$», los siguientes caracteres del identificador pueden ser letras o dígitos.

2.4. Palabras reservadas

El conjunto de palabras reservadas en Java es el que aparece en la tabla Tabla 2-1.

Tabla 2-1. El conjunto de palabras reservadas de Java

abstract	continue	for	new	switch
boolean	default	goto	null	synchronized
break	do	if	package	this
byte	double	implements	private	threadsafe
byvalue	else	import	protected	throw
case	extends	instanceof	public	transient
catch	false	int	return	true

char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	while

Ningún identificador puede llevar el nombre de una palabra reservada.

2.5. Separadores

En Java existen seis separadores distintos. A continuación se muestra el uso de cada uno de ellos.

Los paréntesis ():

Delimitan listas de parámetros.
Modifican la precedencia de una expresión.
Delimitan condiciones.
Indican el tipo en las coerciones.

Las llaves {}:

Definen bloques de código.
Delimitan las lista de valores iniciales de los arrays.

Los corchetes []:

Declaran vectores y permiten acceder a sus elementos.

El punto y coma «;»:

Terminan instrucciones.

La coma «,»:

Separan identificadores en declaraciones.
Encadenan expresiones.

El punto «.»:

Acceden a los atributos y métodos de una clase.

Acceden a un subpaquete de un paquete.

Capítulo 3. Tipos de datos en Java.

En este capítulo se presentan los tipos de datos que se pueden utilizar en Java. En Java existen dos grupos de tipos de datos, tipos primitivos y tipos referencia. Los tipos de datos primitivos son los mismo que en C/C++, `int, float, double` etcétera; los tipos referencias sirven para acceder a los atributos y métodos de los objetos. En este capítulo también se presentan los operadores definidos en el lenguaje, como crear arrays y la clase para contener cadenas /`String`).

3.1. Tipos de datos primitivos.

En Java existen además de objetos tipos de datos primitivos (`int, float,` etcétera). Al contrario que en C o C++ en Java el tamaño de los tipos primitivos no depende del sistema operativo o de la arquitectura, en todas las arquitecturas y bajo todos los sistemas operativos el tamaño en memoria es el mismo. En la Tabla 3-1 se muestran estos tamaños.

Tabla 3-1. Los tipos de datos primitivos en Java.

Tipo	Definición
boolean	`true o false`
char	Carácter Unicode de 16 bits
byte	Entero en complemento a dos con signo de 8 bits
short	Entero en complemento a dos con signo de 16 bits
int	Entero en complemento a dos con signo de 32 bits
long	Entero en complemento a dos con signo de 64 bits
float	Real en punto flotante según la norma IEEE 754 de 32 bits
double	Real en punto flotante según la norma IEEE 754 de 64 bits

Es posible *recubrir* los tipos primitivos para tratarlos como cualquier otro objeto en Java. Así por ejemplo existe una clase envoltura del tipo primitivo `int` llamado `Integer`. La utilidad de estas clases envoltura quedará clara cuando veamos las clases contenedoras de datos.

3.1.1. Conversión de tipos.

La conversión o promoción automática de tipos se puede dar en dos casos:

- Al evaluar un expresión los tipos más «pequeños» promocionan al mayor tipo presente en la expresión.
- Si a una variable de un tipo se le asigna un valor de un tipo más «pequeño», este promociona al tipo de la variable a la que se asigna.

Siempre se puede forzar el cambio de tipo mediante el uso de *casting*.

3.2. Tipos de datos referencia.

En Java los objetos, instancias de clases, se manejan a través de referencias. Cuando se crea una nueva instancia de una clase con el operador `new` este devuelve una referencia al tipo de la clase. Para aclararlo veamos un ejemplo:

```
Punto unPunto = new Punto();
```

El operador `new()` reserva espacio en memoria para contener un objeto del tipo `Punto` y devuelve una referencia que se asigna a `unPunto`. A partir de aquí accedemos al objeto a través de su referencia. Es posible, por tanto, tener varias referencias al mismo objeto. Presta atención al siguiente fragmento de código.

```
Punto unPunto = new Punto();
unPunto.print();
Punto otroPunto = unPunto;
otroPunto.setX(1.0f);
otroPunto.setY(2.0f);
unPunto.print();
```

La salida por pantalla es:

```
Coordenadas del punto (0.0,0.0)
Coordenadas del punto (1.0,2.0)
```

Como las dos referencias hacen referencia a la misma instancia, los cambios sobre el objeto se pueden realizar a través de cualquiera de ellas.

3.3. Recolector de basura.

Los objetos que dejan de estar referenciados a través de alguna variable no se pueden volver a recuperar. Para que estos objetos desreferenciados no ocupen memoria, un recolector de basura se encarga de «destruirlos» y liberar la memoria que estaban ocupando. Por lo tanta para «destruir» un objeto basta con asignar a su variable referencia el valor `null` como puedes ver en el siguiente ejemplo.

```
Punto unPunto = new Punto(1.0f, 2.0f);
Punto otroPunto = new Punto(1.0f, -1.0f);
unPunto = new Punto(2.0, 2.0f); // El punto (1.0f, 2.0f) se pierde
otroPunto = null; // El punto (1.0f, -1.0f) se pierde
```

3.4. Declaración de variables. Convenciones

En la Sección 3.2 se mostraron algunos ejemplos de declaraciones de variables. Al elegir sus nombre recuerda seguir las convenciones que se dieron en la Sección 1.5

Siempre es aconsejable asignar un valor por defecto en el momento de declaración de una variable. En algunos casos, incluso, se producirá un error durante la compilación si hemos olvidado iniciar alguna variable.

3.5. Ámbito de las variables.

El ámbito de las variables está determinado por el bloque de código donde se declaran y todos los bloques que estén anidados por debajo de este. Presta atención al siguiente fragmento de código:

```
{ // Aquí tengo el bloque más externo
  int entero = 1;
  Punto unPunto = new Punto();
  { // Y aquí tengo el bloque interno
    int entero = 2; // Error ya está declarada
    unPunto = new Punto(1.0f, 1.0f); // Correcto
  }
}
```

3.6. Operadores. Precedencia.

Java tiene prácticamente los mismo operadores que C++. En la Tabla 3-2 se muestran todos los operadores de Java, su precedencia y una pequeña descripción.

Tabla 3-2. Los operadores de Java

Precedencia	Operador	Operando	Descripción
1	++, --	Aritmético	Incremento y decremento
1	+, -	Aritmético	Más y menos unarios
1	~	Entero	Complemento de bits
1	!	Booleano	Complemento booleano
1	(tipo)	Cualquiera	Coerción
2	*, /, %	Aritmético	Multiplicación, división y resto
3	+, -	Aritmético	Suma y resta
3	+	String	Concatenación
4	<<	Entero	Desplazamiento a la izquierda

Precedencia	Operador	Operando	Descripción
4	>>	Entero	Desplazamiento a la derecha con signo
4	>>>	Entero	Desplazamiento a la derecha sin signo
5	<, <=, > >=	Aritmético	Comparación
5	instanceof	Objeto	Comparación de tipo
6	==, !=	Primitivo	Igualdad y desigualdad
6	==, !=	Objeto	Igualdad y desigualdad de referencias
7	&	Entero	AND sobre bits
7	&	Booleano	AND booleano
8	^	Entero	XOR sobre bits
8	^	Booleano	XOR booleano
9	|	Entero	OR sobre bits
9	|	Booleano	OR booleano
10	&&	Booleano	AND condicional
11	||	Booleano	OR condicional
12	?:	NA	Operador condicional ternario
13	=	Cualquiera	Asignación
13	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	Cualquiera	Asignación tras operación

La precedencia de los operadores se puede forzar haciendo uso de paréntesis.

Ya que en Java no existen punteros tampoco existen los operadores dirección, referencia y `sizeof()` tan familiares a los programadores de C/C++.

3.7. Arrays.

El uso de arrays en Java es distinto al uso en C/C++. En Java los arrays son objetos, instancias de la clase `Array`, la cual dispone de ciertos métodos útiles.

La declaración sigue la misma sintaxis que en C/C++, se debe declarar el tipo base de los elementos del array. El tipo base puede ser un tipo primitivo o un tipo referencia:

```
int arrayDeEnteros[] = null; // Declara un array de enteros
Punto arrayDePuntos[] = null; // Declara un array de referencias a Puntos
```

La creación del array se hace, como con cualquier otro objeto, mediante el uso del operador `new()`:

```
arrayDeEnteros = new int[100]; /* Crea el array con espacio para
```

```
                                  100 enteros */
arrayDePuntos = new Puntos[100]; /* Crea el array con espacio para
                                    100 referencias a Punto */
```

En el primer caso se reserva espacio para contener 100 enteros. En el segundo caso se crea espacio para contener 100 referencias a objetos de la clase `Punto`, pero no se crea cada uno de esos 100 objetos. En el siguiente ejemplo se muestra como se crea cada uno de esos 100 objetos de la clase `Punto` y se asignan a las referencias del array.

```
for(int i = 0; i < 100; i++) arrayDePuntos[i] = new Punto();
```

Los arrays se pueden iniciar en el momento de la creación, como en el siguiente ejemplo:

```
int arrayDeEnteros[] = {1, 2, 3, 4, 5};
Punto arrayDePuntos[] = {new Punto(), new Punto(1.0f, 1.0f)};
```

Los arrays disponen de un atributo llamado `length` que indica el número de elementos que contiene.

También proporcionan un método para copiar partes de un array sobre otro array:

```
System.arraycopy(origen, inicioOrigen, destino,
                 inicioDestino, longitud);
```

La extensión a arrays de más dimensiones es directa:

```
Punto matriz[][] = new Punto[3][3]; // Declaramos una matriz
Punto fila[] = {new Punto(), new Punto(1.0f, 1.0f),
   new Punto(2.0f, 2.0f)}; // Declaramos e iniciamos una fila
matriz[0] = fila[]; // matriz[0] y fila hacen referencia al
                    // mismo array
```

Igual que en le caso de los arrays unidimensionales lo importante es saber que el hecho de declarar un array no crea los objetos que se referenciaran desde las posiciones del array, únicamente se crean esas referencias.

Como cualquier otro tipo válido, un método también puede devolver un array.

Ahora ya podemos entender el significado de la lista de argumentos del método `main(String args[])`, es un array de `String`s donde cada uno de ellos, y empezando por la posición 0, es un argumento pasado en la línea de órdenes.

```
[belfern@anubis Java]$java HolaJava uno dos tres
```

En este caso `args[0]="uno", args[1]="dos", args[2]="tres"`.

3.8. Cadenas de caracteres.

En Java existe una clase para representar y manipular cadenas, la clase `String`. Una vez creado un `String` no se puede modificar. Se pueden crear instancias de una manera abreviada y sobre ellas se puede utilizar el operador de concatenación `+`: [1]

```
String frase = "Esta cadena es una frase "
String larga =frase + "que se puede convertir en una frase larga."
System.out.println(larga);
```

Notas

1. En Java se ha eliminado la sobrecarga de operadores, el único operador sobrecargado es `+`.

Capítulo 4. Estructuras de control en Java.

Las estructuras de control en Java presentan escasas diferencias con respecto a C/C++, no obstante existen diferencias. Recordemos que que se llama programación estructurada al uso correcto de las estructuras de control, que se resume en que toda estructura de control debe tener un único punto de entrada y un único punto de salida.

Al final del capítulo se presenta el uso de la recursividad en Java.

4.1. Estrucutras condicionales.

Dos son las estructuras de control condicionales en Java: bifurcación y selección múltiple.

4.1.1. Bifurcación: `if-else, if-else-if`

Su sintaxis es:

```
if(condicion) {
  instruccion1();
  instruccion2();
  // etc
} else {
  instruccion1();
  instruccion2();
  // etc
}
```

Es necesario que la condición sea una variable o expresión booleana. Si sólo existe una instrucción en el bloque, las llaves no son necesarias. No es necesario que existe un bloque `else`.

Se pueden anidar como en el siguiente ejemplo

```
if(condicion1) {
  bloqueDeInstrucciones();
} else if(condicion2) {
  bloqueDeInstrucciones();
} else {
  bloqueDeInstrucciones();
}
```

4.1.2. Selección múltiple: switch.

Su sintaxis es la siguiente:

```
switch(expresion) {
  case valor1:
    instrucciones();
    break;
  case valor2:
    instrucciones();
    break;
  default:
    instrucciones();
}
```

La expresión ha de ser una variable de tipo entero o una expresión de tipo entero. Cuando se encuentra coincidencia con un `case` se ejecutan las instrucciones a él asociadas hasta encontrar el primer `break`. Si no se encuentra ninguna coincidencia se ejecutan las instrucciones en `default`. La sección `default` es prescindible.

4.2. Estructuras de repetición.

En Java las estructuras de repetición son las mismas que en C/C++. A continuación se detallan y se indican las pequeñas diferencias con respecto a C/C++.

4.2.1. Repetición sobre un rango determinado. `for`

Bucles `for`, su sintaxis es la siguiente:

```
for(iniciación; condición; incremento) {
  // Bloque de instrucciones
}
```

No es necesario que la condición se base exclusivamente en la variable de control del bucle.

En la parte de iniciación se puede declarar una variable de control del bucle cuyo ámbito será el bucle. Tanto en la parte de *iniciación* como de *incremento* se puede incluir varias expresiones separadas por comas, pero nunca en la parte de *condición*. La condición ha de ser una variable booleana o una expresión que se evalue a un valor booleano.

4.2.2. Repeticiones condicionales: `while, do while`.

Su sintaxis y funcionamiento son iguales que en C/C++, en la estructura de control `while` evalúa la condición antes de ejecutar el bloque de la estructura; en la `do...while` se evalúa la condición después de la ejecución del bloque.

```
while(condición) {
  // Bloqe de instrucciones
}

do {
  // Bloque de instrucciones
} while(condición);
```

Igual que en el caso del `for` la condición ha de ser una variable booleana o una expresión que se evalue a un valor booleano.

4.2.3. Uso de `break` y `continue`.

La palabra reservada `break` además de para indicar el fin del bloque de instrucciones en una instrucción de selección múltiple `switch`, sirve para forzar la salida del bloque de una estructura de repetición.. La palabra reservada `continue`, dentro del bloque de una estructura de repetición condicional, sirve para forzar la evaluación de la condición. Observa los dos ejemplos siguientes y la salida que proporcionan por consola:

```
public class Break {
    public static void main(String [] args) {
 for(int i = 0; i < 10; i++) {
     for(int j = 0; j < 10; j++) {
  if(j > i) break;
  System.out.print(j+",");
     }
     System.out.println("");
 }
    }
}
```

```
0,
0,1,
0,1,2,
0,1,2,3,
0,1,2,3,4,
0,1,2,3,4,5,
0,1,2,3,4,5,6,
0,1,2,3,4,5,6,7,
0,1,2,3,4,5,6,7,8,
```

```
0,1,2,3,4,5,6,7,8,9,
```

```
public class Break {
    public static void main(String [] args) {
 for(int i = 0; i < 10; i++) {
     for(int j = 0; j < 10; j++) {
  if(j < i) continue;
  System.out.print(j+",");
     }
     System.out.println("");
 }
    }
}
```

```
0,1,2,3,4,5,6,7,8,9,
1,2,3,4,5,6,7,8,9,
2,3,4,5,6,7,8,9,
3,4,5,6,7,8,9,
4,5,6,7,8,9,
5,6,7,8,9,
6,7,8,9,
7,8,9,
8,9,
9,
```

4.3. Recursividad.

Para que un problema pueda ser resuelto de modo recursivo ha de poseer las siguientes dos características:

- El problema original se ha de poder reducir al mismo problema y con una talla menor.

  Debe existir una caso base o condición de parada.

Por ejemplo, la función factorial se define en los siguientes términos:

- F(n) = n * F(n-1);
- F(0) = 1;

La primera parte de la definición de la función factorial nos dice que se puede calcular el factorial de un número multiplicando ese número por el factorial del número menos uno, se ha reducido el problema al

mismo problema pero con una talla menor. La segunda parte de la definición corresponde al caso base, que fuerza la salida de la recursividad.

Java soporta la programación recursiva. A continuación se muestra una posible solición recursiva para el cálculo de la función factorial:

```
public class Factorial {
    public static void main(String args[]) {
 long n = Long.parseLong(args[0]);
 System.out.println("El factorial de " + n + " es: " + Factorial(n));
    }

    private static long Factorial(long n) {
 long resultado;

 if(n > 0) resultado = n * Factorial(n-1);
 else resultado = 1;

 return resultado;
    }
}
```

4.4. Ejercicios

1. Escribe un programa en Java para calcular la función de Fibonacci, definida del siguiente modo:
   - F(n) = F(n-1) + F(n-2)
   - F(0) = 0, F(1) = 1

Capítulo 5. Clases en Java.

La programación orientada a objetos (POO) abstrae las entidades del mundo real como objetos y las relaciones entre ellos como paso de mensajes. Los objetos son instancias o ejemplares de una clase o plantilla y poseen como características atributos (valores) y métodos (acciones).

Java es un lenguaje de programación orientado a objetos. Todo en Java, con la excepción de los tipos primitivos, es un objeto. En este Capítulo se muestra cómo se define una clase y cómo se crean objetos que son instancias de alguna de las clases definidas. También se muestran las diferencias entre el paso por valor y el paso por referencia al llamar a los métodos de una clase. El Capítulo acaba con la sobrecarga de métodos y el método de finalización.

5.1. Definición de una clase en Java.

Cuando creamos una nueva clase que no extiende a ninguna otra, implícitamente se están heredando las propiedades (atributos) y el comportamiento (métodos) de la clase raíz `Object`. Y esto es muy importante ya que se heredan métodos muy interesantes tales como `toString()`, método al que se llama siempre que se quiere mostrar información de una clase por pantalla con `System.out.println()`.

Otros métodos muy interesantes que se implementan son `wait()`, `notify` y `notifyAll()` relacionados con los bloqueos cuando se utiliza programación concurrente. Los *hilos* y su programación son parte fundamental de Java, pero quedan fuera del alcance de esta introducción.

Una clase en Java agrupa un conjunto de atributos y un conjunto de métodos bajo un nombre común. Además, en el momento de la declaración se debe especificar desde donde se puede acceder a la nueva clase. En el Ejemplo 2-1 se muestra la declaración de la clase `Punto`.

5.2. Atributos. Atributos estáticos o de clase.

Es muy importante dar un valor por defecto a los atributos, de lo contrario, Java asignará un valor inicial dependiendo del tipo del atributo. [1]

La definición de cada atributo debe empezar con un modificador de acceso. Los modificadores de acceso indican la visibilidad, es decir, si se puede tener acceso sólo desde la clase (`private`), desde la clase y las clases que heredan de ella (`protected`), desde cualquier clase definida en el mismo paquete (ausencia de modificador) o desde cualquier clase (`public`).

Tras el modificador de acceso se escribe el tipo del argumento, este puede ser un tipo primitivo a de tipo referencia.

Tras el modificador de acceso, un atributo se puede declarar como `static`. Esto implica que no existe una copia de este atributo en cada instancia de la clase, si no que existe uno único común a todas las instancias. A los atributos `static` también se les llama atributos de clase.

Un uso común de los atributos de clase es llevar la cuenta de las instancias creadas para un cierta clase, como se puede ver en el siguiente Ejemplo 5-1

Ejemplo 5-1. Uso de atributos y métodos `static`

```
class UnaClase {
  private static int numeroInstancias = 0; // Atributo de clase

  public UnaClase () {
    numeroInstancias++; // Cada vez que creo una instancia
                        // incremento el contador
  }

  public static int cuantasInstancias() {
    return numeroInstancias;
  }
}
```

Otro modificador que puede afectar al comportamiento de los atributos de una clase es `final`. Si un atributo se declara como `final`, implica que no se puede cambiar su valor una vez definido. Un ejemplo de uso de este modificador son las constantes de clase:

```
public static final float E = 2.8182f;
```

5.3. Métodos. Métodos estáticos o de clase.

Los métodos especifican el comportamiento de la clase y sus instancias. Los modificadores de acceso y su significado son los mismos que al operar sobre atributos. En particular, al declarar un método estático implica que es un método de la clase, y por lo tanto no es necesario crear ninguna instancia de la clase para poder llamarlo. Fíjate que ya conoces un atributo estático `out` en `System.out.println()`.

El conjunto de los métodos públicos de una clase forma su interfase.

Un método declarado `final` implica que no se puede redefinir en ninguna clase que herede de esta.

En el momento de la declaración hay que indicar cual es el tipo del parámetro que devolverá el método o `void` en caso de que no devuelva nada.

También se ha de especificar el tipo y nombre de cada uno de los argumentos del método entre paréntesis. Si un método no tiene argumentos el paréntesis queda vacío, no es necesario escribir `void` El tipo y número de estos argumentos identifican al método, ya que varios métodos pueden tener el mismo nombre, con independencia del tipo devuelto, y se distinguirán entre si por el número y tipo de sus argumentos. Utilizar un mismo nombre para varios métodos es lo que se conoce como sobrecarga de métodos. En la Sección 5.6 verás ejemplos de uso. Los argumentos de los métodos son variable locales a los métodos y existen sólo en el interior de estos. Los argumentos pueden tener el mismo nombre que los atributos de la clase, de ser así, los argumentos «ocultan» a los atributos. Para acceder a los atributos en caso de ocultación se referencian a partir del objeto actual `this`. Vuelve al Ejemplo 2-1 y observa el construtor con parámetros.

Un método declarado `static` pertenece a la clase que lo implementa, por lo tanto no es necesario crear ningún objeto para poder acceder al método, se puede acceder directamente a través del nombre de la clase. Los métodos declarados `static` tienen una restricción muy fuerte, y es que únicamente pueden acceder a atributos o métodos también declarados como `static`. El por qué es muy sencillo, si no fuese así, desde un método `static` ¿Al atributo de qué instancia nos estamos refiriendo, si ni siquiera es necesario que tengamos una?. Un uso frecuente de los métodos declarados `static` es acceder a los atributos `static` de la clase. Copia, compila y ejecuta el siguiente Ejemplo 5-2.

Ejemplo 5-2. Atributos y Métodos estáticos.

```
class UnaClase {
  private static int numeroInstancias = 0; // Atributo de clase

  public UnaClase () {
    numeroInstancias++; // Cada vez que creo una instancia
                        // incremento el contador
  }

  public static int cuantasInstancias() {
    return numeroInstancias;
  }

    public static void main (String args[]) {
      System.out.println(UnaClase.cuantasInstancias());
      UnaClase uc = new UnaClase();
      System.out.println(uc.cuantasInstancias());
    }
}
```

En programación orientada a objetos las llamadas a los métodos se le llama paso de mensajes, llamar a un método es análogo a pasarle un mensaje.

5.4. Creación de objetos

Los métodos que tienen el mismo nombre que la clase tienen un comportamiento especial, sirven para crear las instancias de la clase y se les llama *constructores* De nuevo, en el Ejemplo 2-1 puedes ver que la clase contiene dos constructores, uno de ellos sin parámetros, al que se conoce como constructor por defecto, y otro de ellos con paraámetros; este es un caso particular de la sobrecarga de métodos, varios métodos con el mismo nombre que se diferencian por el número y tipo de sus parámetros.

Si no se define explícitamente un constructor por defecto, Java lo hará por nosotros ya que siempre es necesario que exista.

Si es necesario que un constructor llame a otro constructor lo debe hacer antes que cualquier otra cosa.

Un constructor nunca puede ser `abstract, synchronized` o `final`.

Las clases pueden contener bloques de código que no estén asociados a ninguna clase. Al crear una instancia se ejecuta el código contenido en el bloque y después se ejecuta el constructor adecuado. Estos bloques se pueden declarar `static`, en este caso los bloques se utilizan para iniciar las variables declaradas como `static`. Observa el siguiente Ejemplo 5-3, en el se declara un array `static` para contener los valores del seno entre 0 y 359 grados. En el bloque `static` se inicia el valor de las posiciones del array.

Ejemplo 5-3. Ejemplo de uso de un bloque `static`

```
public class bloqueStatic {
  static double tablaSeno[] = new double[360];
  static {
    for(int i = 0; i < 360; i++)
      tablaSeno[i] = Math.sin(Math.PI*i/180.0);
  }

  public static void main (String args[]) {
    for(int i = 0; i < 360; i++)
      System.out.println("Angulo: "+i+" Seno= "+tablaSeno[i]);
  }
}
```

La creación de objetos se hace con el operador `new`, quien devuelve una referencia al objeto creado. Es esta referencia la que, como ya sabemos, nos sirve para acceder a los atributos y métodos del objeto.

```
Punto unPunto = new Punto(1.0f, 2.0f);
```

5.5. Paso por valor y paso por referencia

En el paso de mensajes o llamada a métodos, el paso de los argumentos se hace por valor, es decir, los argumentos de los métodos se comportan como variables cuyo ámbito es el bloque del método, del tal modo que los valores de la llamada se *copian* sobre esas variables.

Y esto ocurre tanto si los parámetros son de tipo primitivo como referencia. Si los argumentos son de tipo referencia la referencia de llamada se copia sobre la referencia del método, de tal modo que las dos hacen referencia al mismo objeto, y las modificaciones a través de la referencia dentro del método operan sobre el objeto. Pero, si a la referencia dentro del método le asignamos una referencia a otro objeto, al salir del método y desaparecer la referencia dentro del método, las eventuales modificaciones que hayamos hecho sobre el objeto copia desaparecerán. Para aclarar todo esto observa el siguiente ejemplo y su resultado donde utilizamos la clase Punto definida en Ejemplo 2-1:

Ejemplo 5-4. Ejemplo de paso de una referencia.

```
public class pasoPorReferencia {
  private static void cambiaReferencia (Punto puntoLocal) {
    puntoLocal = new Punto(1.0f, 2.0f);
    System.out.println("Estoy dentro del método.");
    puntoLocal.print();
  }

  public static void main(String args[]) {
    Punto unPunto = new Punto();
    System.out.println("Estoy fuera del método.");
    unPunto.print();
    cambiaReferencia(unPunto);
    System.out.println("Estoy fuera del método.");
    unPunto.print();
  }
}
```

```
Estoy fuera del método.
Coordenadas del punto (1.0, 1.0)
Estoy dentro del método.
Coordenadas del punto (1.0, 2.0)
Estoy fuera del método.
Coordenadas del punto (1.0, 1.0)
```

5.6. Sobrecarga de Métodos

La sobrecarga de métodos implica que se definen varios métodos con el mismo nombre. En este caso se diferencian entre ellos por el número y tipo de sus argumentos. Un caso particular es la sobrecarga de constructores. En el Ejemplo 2-1 la clase definida posee dos constructores: `Punto()` y `Punto(float`

`x, float y)`, ambos con el mismo nombre pero el primero de ellos no recibe parámetros y el segundo recibe dos reales de tipo `float`

5.7. Finalización

Antes de que un objeto sea completamente eliminado de la memoria por el *recolector de basura*, se llama a su método `finalize()`. Este método está definido en la clase `Object` de la que hereda implícitamente cualquier nueva clase. El tema de la herencia lo veremos en el Capítulo 6.

Al llamar al método `finalize()` podemos comprobar si nos queda alguna tarea por hacer antes de que el objeto sea destruido, por ejemplo, cerrar los posibles ficheros que tengamos abiertos, cerrar las conexiones de red, etcétera.

5.8. Ejercicios

1. Modifica el Ejemplo 5-3 declarando un método `static` al que se le pase el valor de un ángulo en grados y nos devuelva el seno.

Notas

1. Es aún más importante dar un valor inicial a las variables declaradas en bloques, ya que de no hacerlo, el compilador de Java producirá un error.

Capítulo 6. Herencia e `Interface` en Java

En este capítulo vas a conocer una de las características más importantes de la programación orientada a objetos, la herencia. Mediante el uso de la herencia las clases pueden extender el comportamiento de otra clase permitiendo con ello un gran aprovechamiento del código.

Java no soporta herencia múltiple, que es por otro lado fuente de conflictos en los lenguajes que sí la permiten como C++. Sin embargo, en Java se definen lo que se conoce como `interface` que permiten añadir múltiples características a las clases que los implementan. Una clase en Java solo puede extender una única clase base pero puede implementar varios `interface`.

6.1. Herencia

La herencia induce una jerarquía en forma de árbol sobre las clases con raíz en la clase `Object`. Una clase se dice que hereda o extiende a otra clase antecesora. La palabra reservada `extends` sirve para indicar que una clase extiende a otra. La clase que extiende a otra hereda todos los atributos y métodos de la clase antecesora que no sean privados (`private`). La clase antecesora puede extender a su vez otra clase. En el Ejemplo 6-1 se muestra un ejemplo de herencia.

Ejemplo 6-1. Un ejemplo de herencia

```
public class Punto3D extends Punto {
  protected float z;

  public Punto3D() {
    super();
    this.z = 0.0f;
  }

  public Punto3D(float x, float y) {
    super(x, y);
    this.z = 0.0f;
  }

  public Punto3D(Punto unPunto) {
    super(unPunto.getX(), unPunto.getY());
    this.z = 0.0f;
  }

  public Punto3D(float x, float y, float z) {
    super(x, y);
    this.z = z;
  }

  public float getZ() {
    return z;
  }
```

```
  public Punto inverso() {
    return new Punto3D(-getX(), -getY(), -z);
  }

  public String toString() {
    return "("+getX()+", "+getY()+", "+z+")";
  }

  public void print() {
    System.out.println("Coordenadas del punto "+this);
  }
}
```

En el ejemplo anterior vemos como la clase `Punto3D` extiende a la clase `Punto`, de tal modo que todos los atributos y métodos de la clase `Puntoque` no sean privados serán heredados por la clase `Punto3D` y excepto los declarados en la clase antecesora como `private` serán todos accesibles. Además se añaden nuevos métodos y, como veremos en la Sección 6.1.1, otros se sobrescriben.

En Java no existe herencia múltiple, sólo se puede extender una clase en cada paso de herencia. Como veremos en la Sección 6.2 la herencia múltiple se consigue a través de los `interface`.

Un objeto de una clase puede asignarse a otro de la clase a la que extiende, observa el ejemplo:

```
public class herenciaPunto3D {
  public static void main(String args[]) {
    Punto3D unPunto3D = new Punto3D(1.0f, 2.0f, 3.0f);
    Punto unPunto = unPunto3D;
    unPunto.print();
    unPunto.inverso().print();
  }
}
```

```
Coordenadas del punto (1.0, 2.0, 3.0)
Coordenadas del punto (-1.0, -2.0, -3.0)
```

En el ejemplo anterior se asigna una referencia a un objeto de la clase `Punto3D` a una referencia a un objeto de la clase `Punto`. La inversa no es posible, es decir, no se puede asignar a una referencia otra referencia de un objeto que pertenece a su clase antecesora. De poderse hacer, ¿Cual sería el valor de los atributos que no están definidos en la clase antecesora?.

6.1.1. Sobrescritura de variables y métodos.

Una clase que extiende a otra puede declarar atributos con el mismo nombre que algún atributo de la clase a la que extiende, se dice que el atributo se ha sobrescrito u ocultado. Un uso de sobre escritura de atributos es la extensión del tipo como se puede observa en el siguiente ejemplo, el atributo `dato` en la clase base es de tipo `int` mientras que en la clase que extiende a esta se declara con tipo `float`.

```
class Entero {
  public int dato;
  ...
}

class Real extends Entero {
  public float dato;
  ...
}
```

Como vimos en el Ejemplo 6-1 una clase que extiende a otra puede sobrescribir los métodos de su clase antecesora. Cuando se llame a un método sobrescrito se ejecutará el código asociado a este y no el de la clase que se extiende.

En el Ejemplo 6-1 la clase `Punto3D` extiende a la clase `Punto` y sobrescribe los métodos `inverso()`, `toString()` y `print()`. Cuando se cree un objeto de la clase `Punto3D` conocerá los métodos de la clase base que no se han sobrescrito, los añadidos por la clase extendida y los sobrescritos por la clase base. Si una referencia a este objeto se asigna a una referencia de la clase antecesora y se llama a uno de los métodos que se han sobrescrito, se ejecutaran los de la clase que extiende y no el de la antecesora. Es muy importante que te quede claro este comportamiento.

Finalmente, un método declarado `static` en una clase antecesora puede sobrescribirse en una clase que la extienda, pero también se debe declarar `static`, de lo contrario se producirá un error en tiempo de compilación.

6.1.2. Sobrescritura de constructores.

Como ya sabes los constructores son métodos invocados en el momento de la creación de instancias. Como cualquier otro método se pueden sobrescribir en el momento de la extensión. Si el constructor de una clase que extiende a otra necesita invocar a un constructor de su clase antecesora, lo debe hacer antes de llamar a cualquier otro método. En los constructores de la clase `Punto3D` puedes ver un par de ejemplos.

Cuando se crea una instancia de una clase que extiende a otra, de alguna manera se debe construir una instancia de la clase antecesora. Si no se especifica lo contrario, al crear una instancia de una clase se

invoca al constructor por defecto de la clase antecesora, por lo que este constructor debe estar definido en la clase que se extiende. Observa el siguiente ejemplo:

```
public class A {
  private int a;

  public A(int a) {
    this.a = a;
  }
}

public class B extends A {
  private int b;

  public B(int b) {
    this.b = b;
  }

  public static void main (String args[]) {
    B b = new B(2); // Se produce un error
  }
}
```

Cuando en `main` se intenta crear un objeto de la clase `B`, su constructor intenta llamar al constructor por defecto de la clase que ha extendido `A()`, pero este no existe, por lo que se produce un error. Este error se puede evitar de dos modos, bien definiendo explícitamente un constructor por defecto en la clase antecesora `A`, o bien haciendo una llamada explicita desde el constructor de `B` a un constructor que si exista en `A`. En el siguiente ejemplo se muestran estos dos casos:

```
// Primera solución: Constructor por defecto

public class A {
  private int a;

  public A() { // Definimos el constructor por defecto
    a = 0;
  }

  public A(int a) {
    this.a = a;
  }
}

public class B extends A {
  private int b;

  public B(int b) {
    this.b = b;
  }

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

```
    B b = new B(2); // OK
  }
}

// Segunda solución: Llamada explicita a un constructor
public class A {
  private int a;

  public A(int a) {
    this.a = a;
  }
}

public class B extends A {
  private int b;

  public B(int b) {
    super(0); // Se llama a A(int a)
    this.b = b;
  }

  public static void main (String args[]) {
    B b = new B(2); // OK
  }
}
```

Como habrás observado en el ejemplo anterior, una clase que extiende a otra puede acceder a los métodos de la clase extendida desde un método propio a través de la palabra reservada `super`.

6.1.3. Vinculación dinámica.

Observa el comportamiento del siguiente programa:

```
// A.java
public class A {
    public A() {};
    public void metodo() {
 System.out.println("Clase A");
    }
}

// B.java
class B extends A {
    public B() {};

    public void metodo() {
 System.out.println("Clase B");
    }
}
```

```
// Principal.java
class Principal {
    public static void main(String args[]) {
 System.out.println("Dame un numero:");
 int entero = Teclado.leeEntero();
 A a;
 if(entero > 100)  a = new A();
 else a = new B();
 a.metodo();
    }
}
```

Observa que en el momento de la compilación no sabemos qué método se debe ejecutar, ya que depende del valor de una variable que se le pide al usuario. La decisión se tomará en tiempo de ejecución. Este caso se llama vinculación dinámica, y ocurre cuando una clase hija sobreescribe métodos de la clase padre.

6.1.4. El operador `instanceof`.

El operador `instanceof` nos dice cual es la verdadera naturaleza del objeto al que referencia una variable, con independencia del tipo de la variable referencia. Observa el siguiente ejemplo, basado en las clases del ejemplo anterior:

```
// IstanceOf.java
public class InstanceOf {
    public static void main(String args[]) {
 A a = new A();
 B b = new B();
 System.out.println("Es a instancia de A: "+(a instanceof A));
 System.out.println("Es b instancia de B: "+(b instanceof B));
 System.out.println("Es a instancia de B: "+(a instanceof B));
 System.out.println("Es b instancia de A: "+(b instanceof A));
 a = b;
 System.out.println("-----------------");
 System.out.println("Es a instancia de A: "+(a instanceof A));
 System.out.println("Es b instancia de B: "+(b instanceof B));
 System.out.println("Es a instancia de B: "+(a instanceof B));
 System.out.println("Es b instancia de A: "+(b instanceof A));
    }
}
```

El resultado producido es el siguiente:

```
Es a instancia de A: true
Es b instancia de B: true
Es a instancia de B: false
```

```
Es b instancia de A: true
-----------------
Es a instancia de A: true
Es b instancia de B: true
Es a instancia de B: true
Es b instancia de A: true
```

6.1.5. Clases abstractas.

Un método se puede declarar como `abstract`. El método así declarado puede no implementar nada. Si una clase contiene uno o más métodos declarados como `abstract`, ella a su vez debe ser declarada como `abstract`. No se pueden crear instancias de una clase declarada como `abstract`. Las clases que la extiendan deben *obligatoriamente* sobrescribir los métodos declarados como `abstract`.

Las clases `abstract` funcionan como plantillas para la creación de otras clases.

6.2. Interfaces

Los `interface` son la «generalización» de las clases abstractas. Si queremos que una determinada clase vaya a tener cierto comportamiento, hacemos que implemente un determinado `interface`. En el `interface` no se implementa el comportamiento, únicamente se especifica cual va a ser, es decir, se declaran los métodos pero no se implementan. No se pueden crear intancias de un `interface`.

Siguiendo con nuestro ejemplo de la clase `Punto`, hasta ahora con ella sólo podemos representar puntos matemáticos, esos puntos no se pueden dibujar. Si deseamos que posean la característica de poderse dibujar lo indicamos a través de un `interface`. Definamos este `interface`:

```
interface sePuedeDibujar {
  static final Color rojo = Color.RED;
  void dibujate();
  void ponColor(Color unColor);
  Color dameColor();
}
```

Todos los métodos declarados en un `interface` son públicos, así como sus atributos y el mismo `interface`.

Ahora podemos declarar una nueva clase que además de ser un punto, se puede dibujar:

```
public class PuntoVisible implements sePuedeDibujar {
```

```
  private Color color;
  ...
  public void dibujate() {
    System.out.println("Se dibuja el punto");
  }

  public void ponColor(Color unColor) {
    color = unColor;
  }
  // Y el resto de métodos.
}
```

Es necesario que la clase implemente todos los métodos declarados en el `interface`.

Como también has podido observar en el ejemplo anterior, en un `interface` se pueden declarar constantes (`static final`) pero no otro tipo de atributos.

Una referencia a un `interface` puede referenciar a cualquier instancia de clase que implemente dicho `interface`. Observa el siguiente Ejemplo 6-2 :

Ejemplo 6-2. `interface` que referencia a instancias de clases distintas.

```
public class CirculoVisible implments sePuedeDibujar {
  ...
  public void dibujate() {
    System.out.println("Se dibuja el circulo");
  }
}
...
// En el programa principal
PuntoVisible pv = new PuntoVisible();
CirculoVisible cv = new CirculoVisible();
SePuedeDibujar spd = pv;
spd.dibujate();
spd = cv;
spd.dibujate();
```

La salida por consola es:

```
Se dibuja el punto.
Sedibuja el circulo.
```

Finalmente, un `interface` puede extender o otro u otros `interface`. Y si una clase extiende este interface, debe implementar todos los métodos definidos en todos los `interface`.

6.3. Paquetes.

Los paquetes sirven para organizar jerarquicamente las clases. Para declarar que una clase pertenece a un paquete basta incluir la sentencia `package ejemplo.nombre.paquete;` en el fichero de definición de la clase. El fichero de clase resultado de la compilación se debe colocar dentro de algún directorio que pertenezca al `calsspath` y bajo la estructura de directorios ejemplo/nombre/paquete.

Al estructurar de este modo las clases, estamos estableciendo un dominio de nombres que la máquina virtual de Java utiliza, por ejemplo, cuando nuestra aplicación utiliza clases distribuidas en una red de computadores.

Es recomendable que las clases que estan lógicamente realacionadas pertenezcan al mismo paquete. Por ejemplo, al inicio de los ficheros de definición `Punto.java` y `Punto3D.java` debemos incluir `package matematicas.geometria;`

Todas las clases dentro de un mismo paquete tienen acceso al resto de clases declaradas como públicas. Como ya sabes, para poder acceder a una clase de otro paquete se ha de importar este mediante la sentencia `import`.

Capítulo 7. Excepciones en Java.

En este capítulo se muestra qué es una excepción y como se manejan las excepciones en Java. Además se muestra como crear excepciones propias.

7.1. Qué es una excepción.

Una excepción es una situación anómala a la que llega la ejecución de un programa. Estas situaciones anómalas pueden ser desde el intento de abrir un fichero que no existe, a la división por cero.

Java proporciona un mecanismo para detectar y solucionar las excepciones que se puede llegar a producir durante la ejecución de un programa. En Java estamos obligados a tratar las excepciones cuando se producen, bien gestionándolas directamente o bien desentendiéndonos de ellas, pero hasta esto último debemos hacerlo explicitamente.

7.2. Tipos de excepciones.

En Java existen dos grandes tipos de excepciones: los *Errores* y las *Excepciones* propiamente dichas.

Los *Errores* son situaciones irrecuperables, por ejemplo fallos de la máquina vitual. Ante ellos no hay más alternativa que cerrar la aplicación, y no estamos obligados a gestionarlas.

Las excepciones son situaciones anómales ante las cuales bien debemos reaccionar o bien nos desentendemos explítamente. Cuando una excepción se produce se acompaña de toda la información relevante para que podamos gestionarla.

Un caso particular son las excepciones que derivan de `RuntimeException`, como por ejemplo `NullPointerException`. No estamos obligados a tratar este tipo de excepciones, ya que sería muy incómodo tener que comprobar cada vez que, por ejemplo, una referencia es válida antes de utilizarla. Aunque, si deseamos tratar este tipo de excepciones podemos hacerlo.

7.3. Cómo se gestiona una excepción: `try...catch...finally`.

Toda sentencia susceptible de lanzar una excepción debe ir en un bloque `try{...}`. Si se lanza la excepción, la ejecución se abandonará en la sentencia que la lanzó y se ejecutará el bloque `catch(Exception e) {...}` que atrape la excepción generada. Veamos un ejemplo:

Ejemplo 7-1. Ejemplo de tratamiento de excepciones.

```
try {
  readFromFile("esteFicheroNoExiste");
}
catch(FileNotFoundException e) {
  //Aquí tratamos esta excepción
}
catch(IOException e) {
  //Aquí tratamos esta otra
}
finally {
  //Aquí realizamos las tareas comunes.
}
```

Si el fichero del que se pretende leer no existe, se lanza una excepción del tipo `FileNotFoundException` que atrapamos en el bloque `catch` correspondiente. En la jerarquía de las clases excepción, `FileNotFoundException` hereda de `IOException` que a su vez hereda de `Exception` que es la raíz de esta jerarquía; por lo tanto los bloques `catch` deben aparecer de mayor a menor profundidad en la jerarquía, ya que de lo contrario como un `FileNotFoundException` «es» un `IOException`, la excepción quedaría atrapada en este bloque.

Como hemos comentado antes, no es necesario que tratemos las excepciones, pero si no lo vamos a hacer, debemos indicarlo explicitamente. El modo de hacerlo es indicando que el método dentro del cual se puede lanzar una excepción a su vez la «relanza», aquí tienes un ejemplo:

```
void metodoLanzador(int a) throws IOException, ClassNotFoundException {...}
```

Una situación común es que en un método en el que se pueden producir varias excepciones, sea necesario liberar recursos si alguna de ellas se produce. Piensa por ejemplo en un método que esté accediendo a una base de datos, si se produce alguna excepción probablemente deseemos cerrar la conexión con la base de datos. Para evitar escribir el mismo código sea cual sea la excepción que se produce el par `try{...} catch{...}` se puede ampliar con un bloque `finally{...}`, de modo que, sea cual sea la excepción que se produzca, al salir del bloque `catch{...}` que la trate siempre se ejecuta el bloque `finally`. En el Ejemplo 7-1 se muestra un caso de uso.

7.4. Creación de excepciones propias.

Para crear una excepción propia basta extender la clase `Exception` o la excepción más adecuada, y en el constructor de la clase llamar a la clase padre con el mensaje que se desee mostrar cuando se produzca la excepción.

Para lanzar una excepción explicitamente, utilizamos la palabra reservada `trhow` e indicamos en la declaración del método que puede lanzar la excepción deseada. En el siguiente código tienes un ejemplo. Observa que para lanzar una excepción debes crear un objeto del tipo de la excepción que deseas lanzar.

Ejemplo 7-2. Construcción de una excepción propia.

```
public class MiExcepcion extends Excepcion {
  public MiExcepcion() {
    super("Texto de la excepcion");
  }
}

public class LanzaExcepcion {
  public void metodo() throws MiExcepcion {
    //...
    if(a < b) trhow new MiExcepcion();
    //...
  }
}

public class OtraClase {
  public void metodo() {
    LanzaExcepcion le = new LanzaExcepcion();
    try {
      le.metodo();
    }
    catch (MiExcepcion e) {
      System.err.println(e.getMessage());
    }
  }
}
```

Capítulo 8. Algunas clases de utilidad.

Hasta ahora has aprendido el uso de algunas clases presentes en el nucleo de Java. Existe una gran cantidad de clases en el nucleo de Java las cuales implementan un amplio abanico de posibilidades. Cuantas más de estas clases conozcas, más sencilla te resultará la programación de tus propias aplicaciones ya que podrás reutilizarlas en tu propio código. En este capítulo se presentan las grupos de clases más comunes en programación y se muestra como utilizarlas.

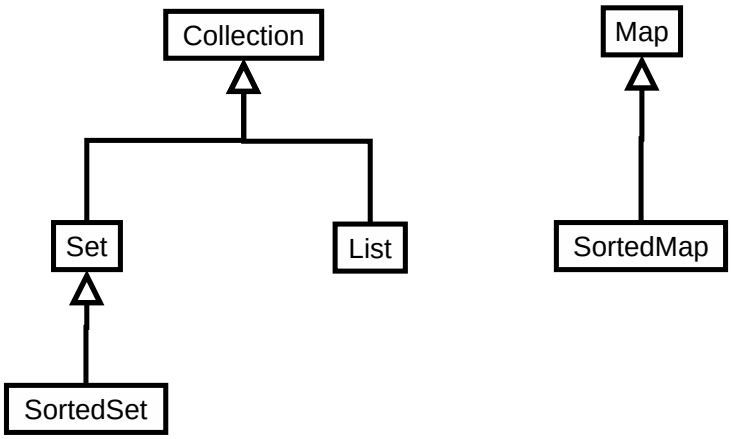
8.1. Colecciones.

En la Figura Figura 8-1 se muestra cual es la jerarquia de las interfaces sobre las que se implementan las colecciones.

La interface `Collection` define las propiedades de un contenedor para objetos. La interface `Map` es un contenedor para objetos que se almacenan junto a una clave. La interface `Collection` tiene dos subinterfaces, `Set` que no permite objetos repetidos, y `List` que mantiene un orden entre los objetos que contiene. A su vez `SortedSet` es un `Set` donde sus elementos están ordenados.

Por otro lado nos encontramos con la interface `Map` que define las propiedades de un contenedor donde los objetos se almacenan junto a una clave. `SortedMap` establece, además, un orden en los elementos almacenados.

Figura 8-1. Jerarquia de los interfaces para colecciones.



Las clases más comunes que implementan a estos interfaces aparecen en la siguiete tabla:

Tabla 8-1. Interfaces y clases colección.

Interface	Implementacion

Interface	Implementacion
Set	HashSet
SortedSet	TreeSet
List	ArrayList, LinkendList, Vector
Map	HasMap, HashTable
SortedMap	TreeMap

La interface `Collection` proporciona los siguientes métodos:

Tabla 8-2. Métodos de la interface `Collection`.

public boolean add(Object o)
public boolean remove(Object o)
public boolean contains(Object o)
public int size()
public boolean isEmpty()
public Iterator iterator()
public Object[] toArray()
public Object[] toArray(Object[] a)

La interface `List` añade además los siguientes métodos:

Tabla 8-3. Métodos de la interface `List`

public void add(int posicion, Object o)
public void remove(int posicion)
public Object get(int posicion)
public void set(int posicion, Object o)

Con respecto a la interface `Map`, implementa los siguientes métodos:

Tabla 8-4. Métodos de la interface `Map`

public Object put(Object clave, Object valor)
public Object get(Object clave)
public Object remove(Object clave)
public int size()
public Set keySet()
public Collection values()

Observa que la interface `Collection` proporciona un iterador. Este iterador itera sobre los elementos de la colección y los devuelve uno a uno. Los métodos que proporciona esta iterface son:

Tabla 8-5. Métodos de la interface `Iterator`

public Object next()
public boolean hasNext()
public void remove()

Hay que tener en cuenta, que la mayoría de los métodos de las colecciones devuelven un objeto de la clase `Object`, que deberemos modelar para obtener un objeto de la clase que deseemos manipular. Por otro lado, tampoco podremos tener coleciones sobre tipos primitivos básicos, para poder utilizar colecciones sobre estos tipos deberemos utilizar las envolturas tal y como veremos en la Seccion Sección 8.4.

A continuación se presenta un sencillo ejemplo de como se utiliza una colección:

```
// A.java
public class A {
    protected int a = 0;
    public A() {};
    public A(int a) { this.a = a; }
    public void getA() {
 System.out.println("Clase A, valor: "+a);
    }
}

// B.java
class B extends A {
    public B() { super(0); }
    public B(int a) { super(a); }
    public void getA() {
 System.out.println("Clase B, valor:"+a);
    }
}

// Coleccion.java
import java.util.ArrayList;
import java.util.Iterator;

public class Coleccion {
    public static void main(String args[]) {
 ArrayList al = new ArrayList();
 for(int i = 0; i < 10; i += 2)
     al.add(new A(i));
 for(int i = 1; i < 10; i += 2)
     al.add(new B(i));

 A a = null;
 Iterator it = al.iterator();
 while(it.hasNext() == true) {
     a = (A)it.next();
     a.getA();
```

```
 }
    }
}
```

Al ejecutar este programa observaras la siguiente salida. Fíjate que aunque en el `ArrayList` hemos añadido objetos de las clases `A` y `B`, realmente se han almacenado como referencias a `Object`, de tal modo que cuando los hemos recuperado con `next()` hemos tenido que hacer un casting.

```
Clase A, valor: 0
Clase A, valor: 2
Clase A, valor: 4
Clase A, valor: 6
Clase A, valor: 8
Clase B, valor:1
Clase B, valor:3
Clase B, valor:5
Clase B, valor:7
Clase B, valor:9
```

8.2. Strings.

La clase `String` sirve para almacenar cadenas de caracteres que no van a modificar su contenido. Los objetos de la clase `String` se pueden crear a partir de constantes de cadena o de arrays de caracteres, y de constantes de tipor primitivos básicos como en el siguiente ejemplo:

```
String cadena = "Esto es una cadena.";
char letras[]={"a", "b", "c"};
String otraCadena = new String(letras);
String pi = String.valueOf(3.141592);
```

Y viceversa, cualquier objeto proporciona una descripción en forma de `String` a través del método `toString()`. De hecho, si quieres que los objetos de tus clases den información relevante cuando se aplica sobre ellos `System.out.println(MiClase o)`, debes sobrecargar el método `public String toString()`.

A continuación se listan algunos métodos de la clase `String`

Tabla 8-6. Métodos de la clase `java.lang.String`

public boolean equals(Object o)
public int length()
public String toLowerCase()

public String toUpperCase()
public String[] split(String s)

Como ya se ha comentado, la clase `String` almacena cadenas de caracteres que no se pueden modificar, cada vez que se concatenan dos objetos de la clase `String` se crea un nuevo objeto de esta clase cuyo contenido es la cadena resultado de la concatenación. Si utilizamos esta clase para realizar este tipo de operaciones, se están creando continuamente objetos de la clase `String`. La clase `StringBuffer` permite manipular cadenas de caracteres, las cadenas almacenadas no son constantes. El método que se debe utiliza para concatenar dos objetos de esta clase es `append(String cadea)`.

8.3. Matemáticas.

Las operaciones aritméticas sobre los tipos primitivos básicos están implementadas directamente sobre Java. Las funciones matemáticas «típicas» están definidas como métodos `static` de la clase `Math` por lo que no es necesario crear ningún objeto de esta clase para poder utilizarlas. A continuación se listan las más comunes:

Tabla 8-7. Métodos de la clase `java.lang.Math`

public static [double|float|etc] abs(double|float|etc)
public static double sin(double a)
public static double cos(double a)
public static double tan(double a)
public static double log(double a)
public static double sqrt(double a)
public static double pow(double a, double b)
public static double asin(double a)
public static double acos(double a)
public static double atan(double a)

Otra clase de utilidad es la que genera una secuencia de números aleatorios, esta es `Random`. En el momento de la creación se le puede pasar una semilla, para que la serie de los números aleatorios generada sea reproducible; si no se proporciona ninguna semilla, se genera una automáticamente basada en el tiempo que devuelve el reloj del sistema. Una vez creado un objeto de esta clase se le puede pedir que devuelva numeros aleatorios de cualquier tipo base según la siguiente tabla:

Tabla 8-8. Métodos de la clase `java.util.Random`

public boolean nextBoolean()
public float nextFloat()
public int nextInt()
public int nextInt(int n)

public long nextLong()
public double nextDouble()
public double nextGaussian()

8.4. Envolturas.

Cada tipo primitiva básico posee una clase envoltura que lo recubre y crea a partir de ellos un objeto. De este modo podemos tratar los tipos primitivos básicos como objetos, y podemos utilizar sobre ellos, por ejemplo, las clases contenedor. La contrapartida es que no se pueden realizar operaciones aritméticas con los objetos de las clases envoltura. La siguiente tabla muestra las clases envoltura que recubren a cada tipo básico:

Tabla 8-9. Clases envoltura.

Tipo base primitivo	Envoltura
void	java.lang.Void
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

Por otro lado, siempre es posible recuperar un elemento del tipo base a partir de un objeto envoltura. En el siguiente listado se muestra un ejemplo.

```
Integer envolturaEntero = new Integer(15);
Float envolturaReal = new Float(15.5f);
int entero = envolturaEntero.intValue();
float real = envolturaEntero.floatValue();
```

Se pueden construir objetos `Integer` a partir de cadenas mediante el método estático `public static Integer valueOf(String s)` al igual que con el resto de envolturas.

Finalmente, se pueden obtener tipos primitivos básicos a partir de una cadena. En el caso de la envoltura `Integer` este método es `public static int parseInt(String s)` o su variante en la que podemos especificar la base en la que se expresa el entero en la cadena `public static int parseInt(String s, int radix)`. A continuación tienes unos ejemplos:

```
parseInt("123", 10); Devuelve el entero 123.
parseInt("100", 2); Devuelve el entero 4.
parseInt("-F", 16); Devuelve el entero -15.
parseInt("Esto no es un entero", 10); //lanza una NumberFormatException.
```

8.5. Ejercicios.

1. Escribir un pograma en Java para jugar al juego de adivinar que número ha «pensado» el ordenador. El número debe estar comprendido entre [1,100] y la partida se ha de almacenar para poder mostrarla completa en caso de que se solicite.
2. Escribir un sencillo programa de agenda telefónica que pida el nombre, dirección y teléfono de cada entrada y lo almacene. El programa ha de tener dos opciones: a) Introducir una nueva entrada, b) Recuperar un dato por el nombre.
3. Escribir un programa en Java que resuelva ecuaciones de segundo grado. Los coeficientes se han de introducir por consola, de tal manera que los argumentos 1 0 -1 representan la ecuación $x^2-1 = 0$.

Capítulo 9. Entrada y salida en Java.

En Java, un flujo es un canal por donde fluyen los datos. Estos datos pueden fluir desde un origen (input) o hacia un destino (output). A su vez, lo que fluye puede ser bytes o caracteres. Java utiliza de un modo homogeneo estos flujos con independencia del origen (teclado, fichero, url, socket, etcétera) y del destino (pantalla, fichero, socket, etcétera).

9.1. Streams en Java.

En Java todas las operaciones de entrada salida se llevan a cabo mediante flujos (streams). Un flujo se contecta a un surtidor de datos, entonces hablaremos de flujos de entrada, o a un consumidor de datos, entonces hablaremos de flujos de salida. Un surtidor típico de datos es el teclado, a él podremos conectar un flujo de entrada por donde nos llegarán datos. De igual modo, un consumidor típico de datos es la pantalla, a ella podremos conectar un flujo de salida hacia donde enviaremos datos.

9.2. Streams de bytes.

En las primeras versiones de Java los únicos flujos de datos disponibles eran de bytes. De hecho, tanto el teclado como la pantalla son flujos de bytes. Veamos los flujos de bytes existentes.

- `InputStream` y `OutputStream`: Son clases abstractas para leer y escribir respectivamente flujos de bytes de datos. Al ser abstractas no se pueden instanciar directamente. Su propósito es servir de base para cualquier otro tipo de flujo de bytes.
- `DataInpuntStrem` y `DataOutputStream`: Se construyen sobre un `InputStream` o un `OutputStream` y sirven para leer o escribir tipos de datos primitivos como `int` o `float`.
- `BufferedInputStream` y `BufferedOutputStream`: Se construyen sobre un `InputStream` o un `OutputStream` y sirven para leer o escribir a través de un buffer, de modo que se optimizan estos procesos.

Existen otros tipos de flujos de bytes que no vamos a tratar.

9.3. Streams de caracteres.

Los flujos de caracteres se incorporaron a Java para facilitar las operaciones de lectura y escritura de caracteres. Veamos las principales clases de flujos de caracteres:

- `Reader` y `Writer`: Son clases abstractas sobre las que se construyen el resto de flujos de caracteres.
- `BufferedReader` y `BufferedWriter`: Se construyen a partir de las anteriores y sirven para leer flujos de caracteres con un buffer.

9.4. Conexión entre streams de bytes y de caracteres

Como ya sabes, el teclado es un flujo de bytes de entrada, por lo que no podemos leer caracteres directamente desde él. De igual modo, la pantalla es un flujo de bytes de salida por lo que tampoco podemos escribir directamente caracteres hacia él. Por ello, existen dos clases que crean flujos de caracteres a partir de flujos de bytes.

`InputStreamReader` se construye sobre un `InputStream` y devulve un `Reader`. De modo análogo, `OutputStreamWriter` se construye sobre un `OutputStream` y devuelve un `Writer`.

Veamos como caso práctico como se leen datos desde el teclado:

```
// Construyo un Reader a partir de System.in
InputStreamReader isr = new InputStreamReader(System.in);
// Lo recubro para utilizar buffer
BufferedReader br = new BufferedReader(isr);
try{
  String cadena = br.readLine(); // Leo una cadena de caracteres
}
catch(IOException e) {System.err.println(e.getMessage);}
```

9.5. Ficheros y streams a ficheros.

9.5.1. El sistema de ficheros.

La clase `File` proporciona acceso al sistema de ficheros, para poder crear o borrar ficheros y directorios, comprobar la existencia de un determinado fichero, etcétera, pero no proporciona acceso al contenido de los ficheros.

Existen varios constructores para crear un objeto de la clase `File`:

```
File fichero = new File("/direccion/absoluta/al/fichero");
File fichero2 = new File("direccionRelativaALaJVM");
File directorio = new File("/directorio");
File fichero3 = new File(directorio, "nombreFichero");
```

Debemos llevar cuidado con el tipo de separador que utliza el sistema operativo sobre el que se ejecuta la JVM. La variable estática `separator` devuelve un `String` con el separador del sistema sobre el que se

está ejecutando la JVM. La variable estática `File.separatorChar` devuelve la misma información pero como un `char`.

Algunos métodos útiles que nos proporciona la clase `File` son: `createNewFile()` crea un nuevo fichero; `delete()` borra un fichero; `exists()` devuelve `true` si existe, `false` en caso contrario; `getAbsolutePath()` devuelve la dirección absoluta del fichero; `isDirectory()` devuelve `true` si el objeto `File` es un directorio, `false` en caso contrario; `isFile()` igual que el anterior pero con fichero.

9.5.2. Streams a ficheros.

Los streams a ficheros se construyen sobre un objeto de la clase `File` y proporcionan acceso al contenido del fichero. Como cabe esperar, existen streams a ficheros para leer y escribir bytes `FileInputStream` y `FileOutputStream` respectivamente, y para leer y escribir caracteres `FileReader` y `FileWriter` respectivamente. Tanto los streams de bytes como de caracteres podemos recubrirlos con streams que añaden características, como por ejemplo lectura o escritura a través de un buffer. Veámoslo con un ejemplo:

```
File f = new File("ficheros.java");
FileReader fr = new FileReader(f);
BufferedReader br = new BufferedReader(fr);
String linea =br.readLine();
```

9.6. Ejercicios

1. Escribir una aplicación en Java que a partir de los datos almacenados en un fichero de texto calcule el centroide y el radio de una nuve de puntos. Cada línea del fichero de datos es una entrada de la forma x y z de cada punto.
2. Escribir una aplicación en Java que lea un fichero de tipo texto donde cada línea representa un dato con la estructura Nombre Apellidos, DNI, y que calcule la letra del NIF y escriba un nuevo fichero donde cada línea representa un dato con la estructura Nombre Apellidos, NIF.