REPASO DE PROGRAMACIÓN EN JAVA

Dpto. Informática y comunicaciones.
Curso. 2024 / 2025
José Luís Casado Valero

Virrey Morcillo
Instituto de Educación Secundaria

1 Las variables e identificadores.

Una variable es una zona en la memoria del computador con un valor que puede ser almacenado para ser usado más tarde en el programa. Las variables vienen determinadas por:

- ***Nombre**, que permite al programa acceder al valor que contiene en memoria. Debe ser un identificador válido.*
- ***Tipo** de dato, que especifica qué clase de información guarda la variable en esa zona de memoria*
- ***Rango** de valores que puede admitir dicha variable.*

1.1 Identificadores.

*Un identificador en Java es una secuencia ilimitada sin espacios de letras y dígitos Unicode, de forma que el primer símbolo de la secuencia debe ser una **letra**, un símbolo de **subrayado** (_) o el símbolo **dólar** ($). Por ejemplo, son válidos los siguientes identificadores:*

x5 *NUM_MAX* *nuevoUsuario* *_siguiente*

RECOMENDACIÓN

Una buena práctica de programación es seleccionar nombres adecuados para las variables, eso ayuda a que el programa se auto-documente, y evita un número excesivo de comentarios para aclarar el código.

1.2 Recomendaciones:

Identificador	Convención	Ejemplo
nombre de variable	Comienza por letra minúscula, y si tienen más de una palabra se colocan juntas y el resto comenzando por mayúsculas	`numAlumnos, suma`
nombre de constante	En letras mayúsculas, separando las palabras con el guión bajo, por convenio el guión bajo no se utiliza en ningún otro sitio	`TAM_MAX, PI`
nombre de una clase	Comienza por letra mayúscula	`String, MiTipo`
nombre de función	Comienza con letra minúscula	`modifica_Valor, obtiene_Valor`

1.3 Palabras reservadas.

Abstract	continue	for	new	switch
assert	default	goto	package s	ynchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

1.4 Tipos de datos primitivos.

TIPOS DE DATOS PRIMITIVOS				
Tipo	**Descripción**	**Bytes**	**Rango**	**Valor por default**
byte	Entero muy corto	1	-128 a 127	0
short	Entero corto	2	-32,768 a 32,767	0
int	Entero	4	-2,147,483,648 a 2,147,483,647	0
long	Entero largo	8	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	0L
float	Numero con punto flotante de precisión individual con hasta 7 dígitos significativos	4	+/-1.4E-45 (+/-1.4 times 10^{-45}) a +/-3.4E38 (+/-3.4 times 10^{38})	0.0f
double	Numero con punto flotante de precisión doble con hasta 16 dígitos significativos	8	+/-4.9E-324 (+/-4.9 times 10^{-324}) a +/-1.7E308 (+/-1.7 times 10^{308})	0.0d
char	Carácter Unicode 2	\u0000 a \uFFFF	'\u0000'	
boolean	Valor Verdadero o Falso	1	true o false	false

1.5 Declaración e inicialización.

Como mínimo debemos indicar el tipo de la variable y el nombre.

Las variables se pueden declarar en cualquier bloque de código, dentro de llaves. Y lo hacemos indicando su identificador y el tipo de dato, separadas por comas si vamos a declarar varias a la vez, por ejemplo:

```
int numAlumnos = 15;
double radio = 3.14, importe = 102.95;
```

De esta forma, estamos declarando **numAlumnos** como una variable de tipo **int**, y otras dos variables **radio** e **importe** de tipo **double**. Aunque no es obligatorio, hemos aprovechado la declaración de las variables para inicializarlas a los valores 15, 3.14 y 102.95 respectivamente.

Si la variable va a permanecer inalterable a lo largo del programa, la declararemos como **constante**, utilizando la palabra reservada **final** de la siguiente forma:

```
final double PI = 3.1415926536;
```

- Las **variables miembro (variables de clase)** sí se inicializan automáticamente, si no les damos un valor. Cuando son de tipo numérico, se inicializan por defecto a 0, si don de tipo carácter, se inicializan al carácter **null** (\0), si son de tipo boolean se les asigna el valor por defecto **false**, y si son tipo referenciado se inicializan a **null**.

- Las **variables locales (variables de función)** no se inicializan automáticamente. Debemos asignarles nosotros un valor antes de ser usadas, ya que si el compilador detecta que la variable se usa antes de que se le asigne un valor, produce un error. Por ejemplo en este caso:

```
int p;
int q = p; // error
```

1.6 Tipos referenciados.

A partir de los ocho tipos datos primitivos, se pueden construir otros tipos de datos. Estos tipos de datos se llaman **tipos referenciados** o **referencias**, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador.

```
int[] arrayDeEnteros;
Cuenta cuentaCliente;
```

Clase String.

Además de los ocho tipos de datos primitivos que ya hemos descrito, Java proporciona un tratamiento especial a los textos o cadenas de caracteres mediante el tipo de dato String. Java crea automáticamente un nuevo objeto de tipo String cuando se encuentra una cadena de caracteres encerrada entre comillas dobles. En realidad se trata de objetos, y por tanto son tipos referenciados, pero se pueden utilizar de forma sencilla como si fueran variables de tipos primitivos:

```
String mensaje;
mensaje= "El primer programa"
```

1.7 Tipos enumerados

Los **tipos de datos enumerados** son una forma de declarar una variable con un conjunto restringido de valores. Por ejemplo, los días de la semana, las estaciones del año, los meses, etc. Es como si definiéramos nuestro propio tipo de datos.

La forma de declararlos es con la palabra reservada **enum**, seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. A los valores que se colocan dentro de las llaves se les considera como constantes, van separados por comas y deben ser valores únicos.

La lista de valores se coloca entre llaves, porque un tipo de datos **enum** no es otra cosa que una especie de clase en Java, y todas las clases llevan su contenido entre llaves.

```
10  public class tiposenumerados {
11      public enum Dias {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
12
13      public static void main(String[] args) {
14          // codigo de la aplicacion
15          Dias diaactual = Dias.Martes;
16          Dias diasiguiente = Dias.Miercoles;
17
18          System.out.print("Hoy es: ");
19          System.out.println(diaactual);
20          System.out.println("Mañana\nes\n"+diasiguiente);
21
22      } // fin main
23
24  } // fin tiposenumerados
```

1.8 Salida estándar por pantalla / Comentarios.

Para mostrar por pantalla un mensaje utilizamos **System.out**, conocido como la salida estándar del programa. Este método lo que hace es escribir un conjunto de caracteres a través de la línea de comandos. En Netbeans esta línea de comandos aparece en la parte inferior de la pantalla. Podemos utilizar **System.out.print** o **System.out.println**. En el segundo caso lo que hace el método es que justo después de escribir el mensaje, sitúa el cursor al principio de la línea siguiente.

El texto en color gris que aparece entre caracteres // son comentarios que permiten documentar el código, pero no son tenidos en cuenta por el compilador y, por tanto, **no afectan a la ejecución del programa**.

1.9 Literales de los tipos primitivos.

Un **literal**, valor literal o constante literal es un valor concreto para los tipos de datos primitivos del lenguaje, el tipo String o el tipo null. Los literales booleanos tienen dos únicos valores que puede aceptar el tipo:

true y **false**. Por ejemplo, con la instrucción **boolean** encontrado = **true**; estamos declarando una variable de tipo booleana a la cual le asignamos el valor literal **true**.

Los **literales enteros** se pueden representar en tres notaciones:

Decimal: por ejemplo 20. Es la forma más común.

Octal: por ejemplo 024. Un número en octal siempre empieza por cero, seguido de dígitos octales (del **0** al **7**).

Hexadecimal: por ejemplo 0x14. Un número en hexadecimal siempre empieza por **0x** seguido de dígitos hexadecimales (del **0** al **9**, de la '**a**' a la '**f**' o de la '**A**' a la '**F**').

Las constantes literales de tipo **long** se le debe añadir detrás una **l** ó **L**, por ejemplo 873L, si no se considera por defecto de tipo **int**. Se suele utilizar **L** para evitar la confusión de la ele minúscula con 1.

Los **literales reales** o en coma flotante se expresan con coma decimal o en notación científica, o sea, seguidos de un exponente **e** ó **E**. El valor puede finalizarse con una **f** o una **F** para indica el formato **float** o con una **d** o una **D** para indicar el formato **double** (por defecto es **double**). Por ejemplo, podemos representar un mismo literal real de las siguientes formas: **13.2**, **13.2D**, **1.32e1**, **0.132E2**. Otras constantes literales reales son por ejemplo: **.54**, **31.21E-5**, **2.f**, **6.022137e+23f**, **3.141e-9d**.

Un **literal carácter** puede escribirse como un carácter entre comillas simples como 'a', 'ñ', 'Z', 'p', etc. o por su código de la tabla Unicode, anteponiendo la secuencia de escape '\' si el valor lo ponemos en octal o '**\u**' si ponemos el valor en hexadecimal. Por ejemplo, si sabemos que tanto en ASCII como en Unicode, la letra A (mayúscula) es el símbolo número 65, y que 65 en octal es 101 y 41 en hexadecimal, podemos representar esta letra como '**\101**' en octal y '**\u0041**' en hexadecimal. Existen unos caracteres especiales que se representan utilizando secuencias de escape:

Secuencia de escape	Significado	Secuencia de escape	Significado
\b	Retroceso	\r	Retorno de carro
\t	Tabulador	\"	Carácter comillas dobles
\n	Salto de línea	\'	Carácter comillas simples
\f	Salto de página	\\	Barra diagonal

2 Operadores y expresiones.

Los **operadores** llevan a cabo operaciones sobre un conjunto de datos u operandos, representados por literales y/o identificadores. Los operadores pueden ser unarios, binarios o terciarios, según el número de operandos que utilicen sean uno, dos o tres, respectivamente. Los operadores actúan sobre los tipos de datos primitivos y devuelven también un tipo de dato primitivo.
Los operadores se combinan con los literales y/o identificadores para formar expresiones. Una **expresión** es una combinación de operadores y operandos que se evalúa produciendo un único resultado de un tipo determinado.
El resultado de una expresión puede ser usado como parte de otra expresión o en una sentencia o instrucción. Las expresiones, combinadas con algunas palabras reservadas o por sí mismas, forman las llamadas **sentencias** o **instrucciones**.

Por ejemplo, pensemos en la siguiente expresión Java:

```
i + 1
```

Con esta expresión estamos utilizando un operador aritmético para sumarle una cantidad a la variable **i**, pero es necesario indicar al programa qué hacer con el resultado de dicha expresión:

```
suma = i + 1;
```

Que lo almacene en una variable llamada suma, por ejemplo. En este caso ya tendríamos una acción completa, es decir, una sentencia o instrucción.

2.1 Operadores aritmeticos.

Operador	Operación Java	Expresión Java	Resultado
-	Operador unario de cambio de signo	-10	-10
+	Adición	1.2 + 9.3	10.5
-	Sustracción	312.5 – 12.3	300.2
*	Multiplicación	1.7 * 1.2	1.02
/	División (entera o real)	0.5 / 0.2	2.5
%	Resto de la división entera	25 % 3	1

El resultado de las operaciones depende de los operandos:

Tipo de los operandos	Resultado
Un operando de tipo long y ninguno real (float o double)	long
Ningún operando de tipo long ni real (float o double)	int
Al menos un operando de tipo double	double
Al menos un operando de tipo float y ninguno double	float

Operadores unarios incrementales y decrementales.

Tipo operador	Expresión Java	
++ (incremental)	Prefija:	Postfija:
	x=3;	x=3;
	y=++x;	y=x++;
	// x vale 4 e y vale 4	// x vale 4 e y vale 3
--(decremental)	5-- // el resultado es 4	

2.2 Operadores de asignación

El principal operador de esta categoría es el operador asignación =, que permite al programa darle un valor a una variable, y que ya hemos utilizado en varias ocasiones en esta unidad. Además de este operador, Java proporciona otros operadores de asignación combinados con los operadores aritméticos, que permiten abreviar o reducir ciertas expresiones.

Operador	Ejemplo en Java	Expresión equivalente
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2

2.3 Operador condicional

El operador condicional **?:** sirve para evaluar una condición y devolver un resultado en función de si es verdadera o falsa dicha condición. Es el único operador ternario de Java, y como tal, necesita tres operandos para formar una expresión.

El primer operando se sitúa a la izquierda del símbolo de interrogación, y siempre será una expresión booleana, también llamada condición. El siguiente operando se sitúa a la derecha del símbolo de interrogación y antes de los dos puntos, y es el valor que devolverá el operador condicional si la condición es verdadera. El último operando, que aparece después de los dos puntos, es la expresión cuyo resultado se devolverá si la condición evaluada es falsa.

condición ? exp1 : exp2

Por ejemplo, en la expresión:

boolean mayoDeEada = (edad > 17) ? true : false;

Se evalúa la condición de si **edad** es mayor que **17**, en caso afirmativo se devuelve el valor de la variable **true**, y en caso contrario se devuelve el valor de **false**.

2.4 Operadores de relación

Los operadores relacionales se utilizan para comparar datos de tipo primitivo (numérico, carácter y booleano). El resultado se utilizará en otras expresiones o sentencias, que ejecutarán una acción u otra en función de si se cumple o no la relación.

Estas expresiones en Java dan siempre como resultado un valor booleano **true** o **false**. En la tabla siguiente aparecen los operadores relacionales en Java.

Operador	Ejemplo en Java	Significado
==	op1 == op2	op1 igual a op2
!=	op1 != op2	op1 distinto de op2
>	op1 > op2	op1 mayor que op2
<	op1 < op2	op1 menor que op2
>=	op1 >= op2	op1 mayor o igual que op2
<=	op1 <= op2	op1 menor o igual que op2

2.5 Operadores lógicos

Los operadores lógicos realizan operaciones sobre valores booleanos, o resultados de expresiones relacionales, dando como resultado un valor booleano.

Los operadores lógicos los podemos ver en la tabla que se muestra a continuación. Existen ciertos casos en los que el segundo operando de una expresión lógica no se evalúa para ahorrar tiempo de ejecución, porque con el primero ya es suficiente para saber cuál va a ser el resultado de la expresión.

Por ejemplo, en la expresión **a && b** si a es falso, no se sigue comprobando la expresión, puesto que ya se sabe que la condición de que ambos sean verdadero no se va a cumplir. En estos casos es más conveniente colocar el operando más propenso a ser falso en el lado de la izquierda. Igual ocurre con el operador **||**, en cuyo caso es más favorable colocar el operando más propenso a ser verdadero en el lado izquierdo.

Operador	Ejemplo en Java	Significado
!	!op	Devuelve **true** si el operando es **false** y viceversa.
&	op1 & op2	Devuelve **true** si op1 y op2 son **true**
|	op1 | op2	Devuelve **true** si op1 u op2 son **true**
^	op1 ^ op2	Devuelve **true** si sólo uno de los operandos es **true**
&&	op1 && op2	Igual que &, pero si op1 es **false** ya no se evalúa op2
||	op1 || op2	Igual que |, pero si op1 es **true** ya no se evalúa op2

2.6 Operadores de bits

Los operadores a nivel de bits se caracterizan porque realizan operaciones sobre números enteros (o **char**) en su representación binaria, es decir, sobre cada dígito binario.

En la tabla tienes los operadores a nivel de bits que utiliza Java.

Operador	Ejemplo en Java	Significado
~	~op	Realiza el complemento binario de op (invierte el valor de cada bit)
&	op1 & op2	Realiza la operación AND binaria sobre op1 y op2
|	op1 | op2	Realiza la operación OR binaria sobre op1 y op2
^	op1 ^ op2	Realiza la operación OR-exclusivo (XOR) binaria sobre op1 y op2
<<	op1 << op2	Desplaza op2 veces hacia la izquierda los bits de op1
>>	op1 >> op2	Desplaza op2 veces hacia la derecha los bits de op1
>>>	op1 >>> op2	Desplaza op2 (en positivo) veces hacia la derecha los bits de op1

2.7 Trabajo con cadenas

El objeto **String** se corresponde con una secuencia de caracteres entrecomillados, como por ejemplo “hola”. Este literal se puede utilizar en Java como si de un tipo de datos primitivo se tratase, y, como caso especial, no necesita la orden **new** para ser creado.

Para aplicar una operación a una variable de tipo **String**, escribiremos su nombre seguido de la operación, separados por un punto. Entre las principales operaciones que podemos utilizar para trabajar con cadenas de caracteres están las siguientes:

- **Creación**. Como hemos visto en el apartado de literales, podemos crear una variable de tipo **String** simplemente asignándole una cadena de caracteres encerrada entre comillas dobles.
- **Obtención de longitud**. Si necesitamos saber la longitud de un **String**, utilizaremos el método **length()**.
- **Concatenación**. Se utiliza el operador **+** o el método **concat()** para concatenar cadenas de caracteres.
- **Comparación**. El método **equals()** nos devuelve un valor booleano que indica si las cadenas comparadas son o no iguales. El método **equalsIgnoreCase()** hace lo propio, ignorando las mayúsculas de las cadenas a considerar.
- **Obtención de subcadenas**. Podemos obtener cadenas derivadas de una cadena original con el método **substring()**, al cual le debemos indicar el inicio y el fin de la subcadena a obtener.
- **Cambio a mayúsculas/minúsculas**. Los métodos **toUpperCase()** y **toLowerCase()** devuelven una nueva variable que transforma en mayúsculas o minúsculas, respectivamente, la variable inicial.
- **Valueof**. Utilizaremos este método para convertir un tipo de dato primitivo (**int**, **long**, **float**, etc.) a una variable de tipo **String**.

2.8 Precedencia de operadores

El orden de precedencia de los operadores determina la secuencia en que deben realizarse las operaciones cuando en una expresión intervienen operadores de distinto tipo.

Las reglas de precedencia de operadores que utiliza Java coinciden con las reglas de las expresiones del álgebra convencional. Por ejemplo:

- La multiplicación, división y resto de una operación se evalúan primero.
- Si dentro de la misma expresión tengo varias operaciones de este tipo, empezaré evaluándolas de izquierda a derecha.

A la hora de evaluar una expresión es necesario tener en cuenta la asociatividad de los operadores. La asociatividad indica qué operador se evalúa antes, en condiciones de igualdad de precedencia. Los operadores de **asignación**, el operador **condicional** (?:), los operadores **incrementales** (++, --) y el **casting** son asociativos por la derecha. El resto de operadores son asociativos por la izquierda, es decir, que se empiezan a calcular en el mismo orden en el que están escritos: de izquierda a derecha. (Ver tabla siguiente).

REGLA DE ORO: Utilizar siempre paréntesis.

Operador	Tipo	Asociatividad
++ --	Unario, notación postfija	Derecha
++ -- + - (cast) ! ~	Unario, notación prefija	Derecha
* / %	Aritméticos	Izquierda
+ -	Aritméticos	Izquierda
<< >> >>>	Bits	Izquierda
< <= > >=	Relacionales	Izquierda
== !=	Relacionales	Izquierda
&	Lógico, Bits	Izquierda
^	Lógico, Bits	Izquierda
|	Lógico, Bits	Izquierda
&&	Lógico	Izquierda
||	Lógico	Izquierda
?:	Operador condicional	Derecha
= += -= *= /= %=	Asignación	Derecha

3 *Conversión* de tipo.

Las conversiones de tipo se realizan para hacer que el resultado de una expresión sea del tipo que nosotros deseamos, en el ejemplo anterior para hacer que el resultado de la división sea de tipo real y, por ende, tenga decimales. Existen dos tipos de conversiones:

Conversiones automáticas. Cuando a una variable de un tipo se le asigna un valor de otro tipo numérico con menos bits para su representación, se realiza una conversión automática. En ese caso, el valor se dice que es promocionado al tipo más grande (el de la variable), para poder hacer la asignación. También se realizan conversiones automáticas en las operaciones aritméticas, cuando estamos utilizando valores de distinto tipo, el valor más pequeño se promociona al valor más grande, ya que el tipo mayor siempre podrá representar cualquier valor del tipo menor (por ejemplo, de int a long o de float a double).

Conversiones explícitas. Cuando hacemos una conversión de un tipo con más bits a un tipo con menos bits. En estos casos debemos indicar que queremos hacer la conversión de manera expresa, ya que se puede producir una pérdida de datos y hemos de ser conscientes de ello. Este tipo de conversiones se realiza con el **operador cast**. El operador cast es un operador unario que se forma colocando delante del valor a convertir el tipo de dato entre paréntesis. Tiene la misma precedencia que el resto de operadores unarios y se asocia de izquierda a derecha.

Debemos tener en cuenta que **un valor numérico nunca puede ser asignado a una variable de un tipo menor en rango, si no es con una conversión explícita**. Por ejemplo:

```
int a;
byte b;
a = 12;                 // no se realiza conversión alguna
b = 12;                 // se permite porque 12 está dentro
                        // del rango permitido de valores para b
b = a;                  // error, no permitido (incluso aunque
                        // 12 podría almacenarse en un byte)
byte b = (byte) a;      // Correcto, forzamos conversión explícita
```

Tabla de Conversión de Tipos de Datos Primitivos

		Tipo destino							
		boolean	char	byte	short	int	long	float	double
Tipo origen	boolean	-	N	N	N	N	N	N	N
	char	N	-	C	C	CI	CI	CI	CI
	byte	N	C	-	CI	CI	CI	CI	CI
	short	N	C	C	-	CI	CI	CI	CI
	int	N	C	C	C	-	CI	CI*	CI
	long	N	C	C	C	C	-	CI*	CI*
	float	N	C	C	C	C	C	-	CI
	double	N	C	C	C	C	C	C	-

- **N:** Conversión no permitida (un **boolean** no se puede convertir a ningún otro tipo y viceversa).
- **CI:** Conversión implícita o automática. Un asterisco indica que puede haber posible pérdida de datos.
- **C:** Casting de tipos o conversión explícita.
- El asterisco indica que puede haber una posible pérdida de datos, por ejemplo al convertir un número de tipo **int** que usa los 32 bits posibles de la representación, a un tipo **float**, que también usa 32 bits para la representación, pero 8 de los cuales son para el exponente.
- En cualquier caso, las conversiones de números en coma flotante a números enteros siempre necesitarán un **Casting**, y deberemos tener mucho cuidado debido a la pérdida de precisión que ello supone.

4 Comentarios.

Los comentarios son muy importantes a la hora de describir qué hace un determinado programa.

Nos podemos encontrar los siguientes tipos de comentarios:

Comentarios de una sola línea. Utilizaremos el delimitador // para introducir comentarios de sólo una línea.

```
// comentario de una sola línea
```

Comentarios de múltiples líneas. Para introducir este tipo de comentarios, utilizaremos una barra inclinada y un asterisco (/*), al principio del párrafo y un asterisco seguido de una barra inclinada (*/) al final del mismo.

```
/* Esto es un comentario
de varias líneas */
```

Comentarios Javadoc. Utilizaremos los delimitadores /** y */. Al igual que con los comentarios tradicionales, el texto entre estos delimitadores será ignorado por el compilador. Este tipo de comentarios se emplean para generar documentación automática del programa. A través del programa javadoc, incluido en JavaSE, se recogen todos estos comentarios y se llevan a un documento en formato .html.

```
/** Comentario de documentación.
Javadoc extrae los comentarios del código y
genera un archivo html a partir de este tipo de comentarios
*/
```

5 *Fundamentos de la Programación* Orientada a Objetos.

Dentro de las distintas formas de hacer las cosas en programación, distinguimos dos paradigmas fundamentales:

Programación Estructurada, se crean **funciones y procedimientos** que definen las acciones a realizar, y que posteriormente forman los programas.

Programación Orientada a Objetos, considera los programas en términos de **objetos** y todo gira alrededor de ellos.

La Programación Orientada a Objetos es un sistema o conjunto de reglas que nos ayudan a descomponer la aplicación en objetos. A menudo se trata de representar las entidades y objetos que nos encontramos en el mundo real mediante componentes de una aplicación. Es decir, debemos establecer una correspondencia directa entre el espacio del problema y el espacio de la solución.

5.1 Beneficios.

Comprensión. Los conceptos del espacio del problema se hayan reflejados en el código del programa, por lo que la mera lectura del código nos describe la solución del problema en el mundo real.

Modularidad. Facilita la modularidad del código, al estar las definiciones de objetos en módulos o archivos independientes, hace que las aplicaciones estén mejor organizadas y sean más fáciles de entender.

Fácil mantenimiento. Cualquier modificación en las acciones queda automáticamente reflejada en los datos, ya que ambos están estrechamente relacionados. Esto hace que el mantenimiento de las aplicaciones, así como su corrección y modificación sea mucho más fácil. Por otra parte, al estar las aplicaciones mejor organizadas, es más fácil localizar cualquier elemento que se quiera modificar y/o corregir.

Seguridad. La probabilidad de cometer errores se ve reducida, ya que no podemos modificar los datos de un objeto directamente, sino que debemos hacerlo mediante las acciones definidas para ese objeto.

Reusabilidad. Los objetos se definen como entidades reutilizables, es decir, que los programas que trabajan con las mismas estructuras de información, pueden reutilizar las definiciones de objetos empleadas en otros programas, e incluso las acciones definidas sobre ellos. Por ejemplo, podemos crear la definición de un objeto de tipo persona para una aplicación de negocios y deseamos construir a continuación otra aplicación, digamos de educación, en donde utilizamos también personas, no es necesario crear de nuevo el objeto, sino que por medio de la reusabilidad podemos utilizar el tipo de objeto persona previamente definido.

5.2 Características.

Cuando hablamos de Programación Orientada a Objetos, existen una serie de características que se deben cumplir. Cualquier lenguaje de programación orientado a objetos las debe contemplar. Las características más importantes del paradigma de la programación orientada a objetos son:

Abstracción. Es el proceso por el cual definimos las características más importantes de un objeto, sin preocuparnos de cómo se escribirán en el código del programa, simplemente lo definimos de forma general. En la Programación Orientada a Objetos la herramienta más importante para soportar la abstracción es la **clase**. Básicamente, una clase es un tipo de dato que agrupa las características comunes de un conjunto de objetos. Poder ver los objetos del mundo real que deseamos trasladar a nuestros programas, en términos abstractos, resulta de gran utilidad para un buen diseño del software, ya que nos ayuda a comprender mejor el problema y a tener una visión global de todo el conjunto. Por ejemplo, si pensamos en una clase **Vehículo** que agrupa las características comunes de todos ellos, a partir de dicha clase podríamos crear objetos como **Coche** y **Camión**. Entonces se dice que **Vehículo** es una abstracción de **Coche** y de **Camión**.

Modularidad. Una vez que hemos representado el escenario del problema en nuestra aplicación, tenemos como resultado un conjunto de objetos software a utilizar. Este conjunto de objetos se crean a partir de una o varias clases. Cada clase se encuentra en un archivo diferente, por lo que la modularidad nos permite modificar las características de la clase que define un objeto, sin que esto afecte al resto de clases de la aplicación.

Encapsulación. También llamada **"ocultamiento de la información"**. La **encapsulación** o **encapsulamiento** es el mecanismo básico para ocultar la información de las partes internas de un objeto a los demás objetos de la aplicación. Con la encapsulación un objeto puede ocultar la información que contiene al mundo exterior, o bien restringir el acceso a la misma para evitar ser manipulado de forma inadecuada. Por ejemplo, pensemos en un programa con dos objetos, un objeto **Persona** y otro **Coche. Persona** se comunica con el objeto **Coche** para llegar a su destino, utilizando para ello las acciones que Coche tenga definidas como por ejemplo conducir. Es decir, **Persona** utiliza **Coche** pero no sabe cómo funciona internamente, sólo sabe utilizar sus métodos o acciones.

Jerarquía. Mediante esta propiedad podemos definir relaciones de jerarquías entre clases y objetos. Las dos jerarquías más importantes son la jerarquía **"es un"** llamada **generalización** o **especialización** y la jerarquía **"es parte de"**, llamada **agregación**. Conviene detallar algunos aspectos:

- La generalización o especialización, también conocida como **herencia**, permite crear una clase nueva en términos de una clase ya existente (herencia simple) o de varias clases ya existentes (herencia múltiple). Por ejemplo, podemos crear la clase **CochedeCarreras** a partir de la clase **Coche**, y así sólo tendremos que definir las nuevas características que tenga.
- La agregación, también conocida como **inclusión**, permite agrupar objetos relacionados entre sí dentro de una clase. Así, un **Coche** está formado por **Motor**, **Ruedas**, **Frenos** y **Ventanas**. Se dice que **Coche** es una agregación y **Motor**, **Ruedas**, **Frenos** y **Ventanas** son agregados de **Coche**.

Polimorfismo. Esta propiedad indica la capacidad de que varias clases creadas a partir de una antecesora realicen una misma acción de forma diferente. Por ejemplo, pensemos en la clase **Animal** y la acción de expresarse. Nos encontramos que cada tipo de **Animal** puede hacerlo de manera distinta, los **Perros** ladran, los **Gatos** maullan, las **Personas** hablamos, etc. Dicho de otra manera, el polimorfismo indica la posibilidad de tomar un objeto (de tipo **Animal**, por ejemplo), e indicarle que realice la acción de expresarse, esta acción será diferente según el tipo de mamífero del que se trate. Esto se consigue normalmente mediante la implementación de interfaces.

6 Clases y Objetos. Características de los objetos.

Cuando escribimos un programa en un lenguaje orientado a objetos, debemos identificar cada una de las partes del problema con objetos presentes en el mundo real, para luego trasladarlos al modelo computacional que estamos creando.

Pero ¿qué es más concretamente un objeto en Programación Orientada a Objetos? Veámoslo.

Un objeto es un conjunto de datos con las operaciones definidas para ellos. Los objetos tienen un estado y un comportamiento.

Por tanto, estudiando los objetos que están presentes en un problema podemos dar con la solución a dicho problema. Los objetos tienen unas características fundamentales que los distinguen:

Identidad. Es la característica que permite diferenciar un objeto de otro. De esta manera, aunque dos objetos sean exactamente iguales en sus atributos, son distintos entre sí. Puede ser una dirección de memoria, el nombre del objeto o cualquier otro elemento que utilice el lenguaje para distinguirlos. Por ejemplo, dos vehículos que hayan salido de la misma cadena de fabricación y sean iguales aparentemente, son distintos porque tienen un código que los identifica.

Estado. El estado de un objeto viene determinado por una serie de parámetros o atributos que lo describen, y los valores de éstos. Por ejemplo, si tenemos un objeto Coche, el estado estaría definido por atributos como Marca, Modelo, Color, Cilindrada, etc.

Comportamiento. Son las acciones que se pueden realizar sobre el objeto. En otras palabras, son los métodos o procedimientos que realiza el objeto. Siguiendo con el ejemplo del objeto **Coche**, el el comportamiento serían acciones como: arrancar(), parar(), acelerar(), frenar(), etc.

6.1 Propiedades y métodos de los objetos.

Como acabamos de ver todo objeto tiene un estado y un comportamiento. Concretando un poco más, las partes de un objeto son:

Campos, Atributos o Propiedades: Parte del objeto que almacena los datos. También se les denomina **Variables Miembro**. Estos datos pueden ser de cualquier tipo primitivo (boolean, char, int, double, etc) o ser su vez ser otro objeto. Por ejemplo, un objeto de la clase Coche puede tener un objeto de la clase Ruedas.

Métodos o Funciones Miembro: Parte del objeto que lleva a cabo las operaciones sobre los atributos definidos para ese objeto.

La idea principal es que el objeto reúne en una sola entidad los datos y las operaciones, y para acceder a los datos privados del objeto debemos utilizar los métodos que hay definidos para ese objeto.

Interacción entre objetos:
Dentro de un programa los objetos se comunican llamando unos a otros a sus métodos. Los métodos están dentro de los objetos y describen el comportamiento de un objeto cuando recibe una llamada a uno de sus métodos. En otras palabras, cuando un objeto, objeto1**,** quiere actuar sobre otro, objeto2**,** tiene que ejecutar uno de sus métodos. Entonces se dice que el objeto2 recibe un mensaje del objeto1.

6.2 Clases.

Un programa informático se compone de muchos objetos, algunos de los cuales comparten la misma estructura y comportamiento. Si tuviéramos que definir su estructura y comportamiento objeto cada vez que queremos crear un objeto, estaríamos utilizando mucho código redundante. Por ello lo que se hace es crear una **clase**, que es una descripción de un conjunto de objetos que comparten una estructura y un comportamiento común. Y a partir de la clase, se crean tantas "instancias" como necesitemos. Esas instancias son los objetos de la clase.

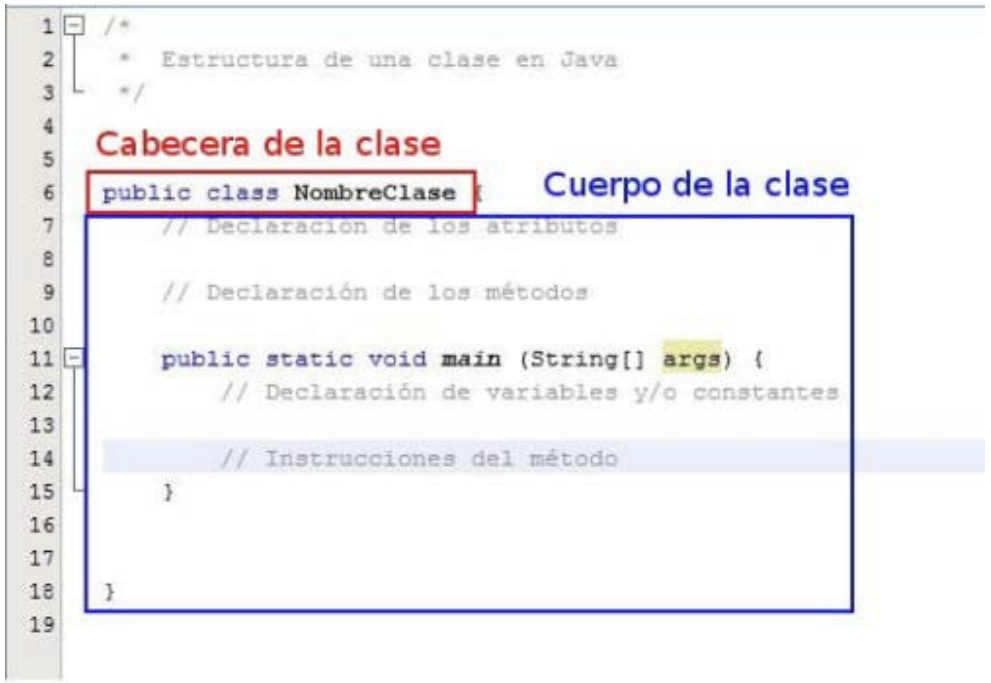
Las clases constan de datos y métodos que resumen las características comunes de un conjunto de objetos. Un programa informático está compuesto por un conjunto de clases, a partir de las cuales se crean objetos que interactúan entre sí.

En otras palabras, **una clase es una plantilla o prototipo donde se especifican**:

Los **atributos** comunes a todos los objetos de la clase.
Los **métodos** que pueden utilizarse para manejar esos objetos.

Para declarar una clase en Java se utiliza la palabra reservada class. La declaración de una clase está compuesta por:

Cabecera de la clase. La cabecera es un poco más compleja que como aquí definimos, pero por ahora sólo nos interesa saber que está compuesta por una serie de modificadores, en este caso hemos puesto public que indica que es una clase pública a la que pueden acceder otras clases del programa, la palabra reservada class y el nombre de la clase.
Cuerpo de la clase. En él se especifican encerrados entre llaves los atributos y los métodos que va a tener la clase.



6.3 Utilización de objetos.

Una vez que hemos creado una clase, podemos crear objetos en nuestro programa a partir de esas clases.

Cuando creamos un objeto a partir de una clase se dice que hemos creado una **"instancia de la clase".** A efectos prácticos, "objeto" e "instancia de clase" son términos similares. Es decir, nos referimos a objetos como instancias cuando queremos hacer hincapié que son de una clase particular.

Los objetos se crean a partir de las clases, y representan casos individuales de éstas.

Para entender mejor el concepto entre un objeto y su clase, piensa en un molde de galletas y las galletas. El molde sería la clase, que define las características del objeto, por ejemplo su forma y tamaño.

Otro ejemplo, imagina una clase Persona que reúna las características comunes de las personas (color de pelo, ojos, peso, altura, etc.) y las acciones que pueden realizar (crecer, dormir, comer, etc.).

Cualquier objeto instanciado de una clase contiene una **copia de todos los atributos** definidos en la clase. En otras palabras, lo que estamos haciendo es reservar un espacio en la memoria del ordenador para guardar sus atributos y métodos. Por tanto, cada objeto tiene una zona de almacenamiento propia donde se guarda toda su información, que será distinta a la de cualquier otro objeto.

En el ejemplo del objeto Persona, las variables instancia serían color_de_pelo, peso, altura, etc. Y los métodos instancia serían crecer(), dormir(), comer(), etc.

6.4 Ciclo de vida de los objetos.

Todo programa en Java parte de una única clase, que como hemos comentado se trata de la clase principal. Esta clase ejecutará el contenido de su método **main()**, el cual será el que utilice las demás clases del programa, cree objetos y lance mensajes a otros objetos.

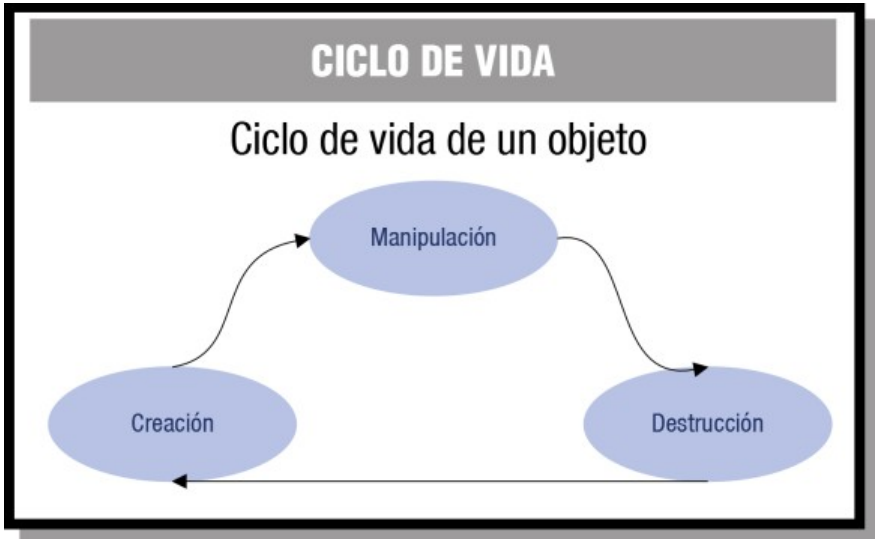
Las instancias u objetos tienen un tiempo de vida determinado. Cuando un objeto no se va a utilizar más en el programa, es destruido por el recolector de basura para liberar recursos que pueden ser reutilizados por otros objetos.

A la vista de lo anterior, podemos concluir que los objetos tienen un ciclo de vida, en el cual podemos distinguir las siguientes fases:

Creación, donde se hace la reserva de memoria e inicialización de atributos.

Manipulación, que se lleva a cabo cuando se hace uso de los atributos y métodos del objeto.

Destrucción, eliminación del objeto y liberación de recursos.



6.5 Declaración.

Para la creación de un objeto hay que seguir los siguientes pasos:

Declaración: Definir el tipo de objeto.

Instanciación: Creación del objeto utilizando el operador new**.**

Pero ¿en qué consisten estos pasos a nivel de programación en Java? Veamos primero cómo declarar un objeto. Para la definición del tipo de objeto debemos emplear la siguiente instrucción:

<tipo> nombre_objeto;

Donde:

- **tipo** es la clase a partir de la cual se va a crear el objeto, y
- **nombre_objeto** es el nombre de la variable referencia con la cual nos referiremos al objeto.

Los tipos referenciados o referencias se utilizan para guardar la dirección de los datos en la memoria del ordenador.

Nada más crear una referencia, ésta se encuentra vacía. Cuando una referencia a un objeto no contiene ninguna instancia se dice que es una referencia nula, es decir, que contiene el valor null**.** Esto quiere decir que la referencia está creada pero que el objeto no está instanciado todavía, por eso la referencia apunta a un objeto inexistente llamado "nulo".

Para entender mejor la declaración de objetos veamos un ejemplo. Veíamos que String realmente este tipo de dato es un **tipo referenciado** y creábamos una variable mensaje de ese tipo de dato de la siguiente forma:

String mensaje;

Nomenclatura: Los nombres de la clase empiezan con mayúscula, como String, y los nombres de los objetos con minúscula, como mensaje, así sabemos qué tipo de elemento utilizando.

String es realmente la clase a partir de la cual creamos nuestro objeto llamado mensaje.

Si observas poco se diferencia esta declaración de las declaraciones de variables que hacíamos para los tipos primitivos. Antes decíamos que mensaje era una variable del tipo de dato String**.** Ahora realmente vemos que mensaje es un objeto de la clase String. Pero mensaje aún no contiene el objeto porque no ha sido instanciado, veamos cómo hacerlo.

Por tanto, cuando creamos un objeto estamos haciendo uso de una variable que almacena la dirección de ese objeto en memoria. Esa variable es una **referencia** o un **tipo de datos referenciado**, porque no contiene el dato si no la posición del dato en la memoria del ordenador.

```
String saludo = new String ("Bienvenido a Java");

String s; //s vale null

s = saludo; //asignación de referencias
```

En las instrucciones anteriores, las variables s y saludo apuntan al mismo objeto de la clase String. Esto implica que cualquier modificación en el objeto saludo modifica también el objeto al que hace referencia la variable s, ya que realmente son el mismo.

6.6 Instanciación.

Una vez creada la referencia al objeto, debemos crear la instancia u objeto que se va a guardar en esa referencia. Para ello utilizamos la orden new con la siguiente sintaxis:

nombre_objeto = new <Constructor_de_la_Clase>([<par1>, <par2>, ..., <parN>]);

Donde:

nombre_objeto es el nombre de la variable referencia con la cual nos referiremos al objeto,

new es el operador para crear el objeto,

Constructor_de_la_Clase es un método especial de la clase, que se llama igual que ella, y se encarga de inicializar el objeto, es decir, de dar unos valores iniciales a sus atributos, y

par1-parN, son parámetros que puede o no necesitar el constructor para dar los valores iniciales a los atributos del objeto

Durante la instanciación del objeto, se reserva memoria suficiente para el objeto. De esta tarea se encarga Java y juega un papel muy importante el **recolector de basura,** que se encarga de eliminar de la memoria los objetos no utilizados para que ésta pueda volver a ser utilizada.

De este modo, para instanciar un objeto String, haríamos lo siguiente:

mensaje = new String();

Así estaríamos instanciando el objeto mensaje. Para ello utilizaríamos el operador new y el constructor de la clase String a la que pertenece el objeto según la declaración que hemos hecho en el apartado anterior. A continuación utilizamos el constructor, que se llama igual que la clase, String.

En el ejemplo anterior el objeto se crearía con la cadena vacía (""), si queremos que tenga un contenido debemos utilizar parámetros en el constructor, así:

mensaje = new String ("El primer programa");

Java permite utilizar la clase String como si de un tipo de dato primitivo se tratara, por eso no hace falta utilizar el operador new para instanciar un objeto de la clase String.

La declaración e instanciación de un objeto puede realizarse en la misma instrucción, así:

String mensaje = new String ("El primer programa");

6.7 Manipulación.

Una vez creado e instanciado el objeto ¿cómo accedemos a su contenido? Para acceder a los atributos y métodos del objeto utilizaremos el nombre del objeto seguido del **operador punto (.)** y el nombre del atributo o método que queremos utilizar. Cuando utilizamos el operador punto se dice que estamos enviando un mensaje al objeto. La forma general de enviar un mensaje a un objeto es:

nombre_objeto.mensaje

Por ejemplo, para acceder a las variables instancia o atributos se utiliza la siguiente sintaxis:

nombre_objeto.atributo

Y para acceder a los métodos o funciones miembro del objeto se utiliza la sintaxis es:

nombre_objeto.método([par1, par2, ..., parN])

En la sentencia anterior par1, par2, etc. son los parámetros que utiliza el método. Aparece entre corchetes para indicar son opcionales.

Para entender mejor cómo se manipulan objetos vamos a utilizar un ejemplo. Para ello necesitamos la Biblioteca de Clases Java o API (Application Programming Interface / Interfaz de programación de aplicaciones). Uno de los paquetes de librerías o bibliotecas es java.awt. Este paquete contiene clases destinadas a la creación de objetos gráficos e imágenes. Vemos por ejemplo cómo crear un rectángulo.

En primer lugar instanciamos el objeto utilizando el método constructor, que se llama igual que el objeto, e indicando los parámetros correspondientes a la posición y a las dimensiones del rectángulo:

Rectangle rect = new Rectangle(50, 50, 150, 150);

Una vez instanciado el objeto rectángulo si queremos cambiar el valor de los atributos utilizamos el operador punto. Por ejemplo, para cambiar la dimensión del rectángulo:

rect.height=100;

rect.width=100;

O bien podemos utilizar un método para hacer lo anterior:

rect.setSize(200, 200);

Código del ejemplo:

```
/*
* Muestra como se manipulan objetos en Java
*/
import java.awt.Rectangle;
/**
*
* @author FMA
*/
public class Manipular {
	public static void main(String[] args) {
		// Instanciamos el objeto rect indicando posicion y dimensiones
		Rectangle rect = new Rectangle( 50, 50, 150, 150 );
		//Consultamos las coordenadas x e y del rectangulo
		System.out.println( "------- Coordenadas esquina superior izqda. -------");
		System.out.println("\tx = " + rect.x + "\n\ty = " + rect.y);
		// Consultamos las dimensiones (altura y anchura) del rectangulo
		System.out.println( "\n----------------- Dimensiones ----------------");
		System.out.println("\tAlto = " + rect.height );
		System.out.println( "\tAncho = " + rect.width);
		//Cambiar coordenadas del rectangulo
		rect.height=100;
		rect.width=100;
		rect.setSize(200, 200);
		System.out.println( "\n-- Nuevos valores de los atributos --");
		System.out.println("\tx = " + rect.x + "\n\ty = " + rect.y);
		System.out.println("\tAlto = " + rect.height );
		System.out.println( "\tAncho = " + rect.width);
	}
}
```

6.8 Destrucción de objetos y liberación de memoria.

Cuando un objeto deja de ser utilizado, es necesario liberar el espacio de memoria y otros recursos que poseía para poder ser reutilizados por el programa. A esta acción se le denomina destrucción del objeto.

En Java la destrucción de objetos corre a cargo del recolector de basura (garbage collector). Es un sistema de destrucción automática de objetos que ya no son utilizados. Lo que se hace es liberar una zona de memoria que había sido reservada previamente mediante el operador new. Esto evita que los programadores tengan que preocuparse de realizar la liberación de memoria.

El recolector de basura se ejecuta en modo segundo plano y de manera muy eficiente para no afectar a la velocidad del programa que se está ejecutando. Lo que hace es que periódicamente va buscando objetos que ya no son referenciados, y cuando encuentra alguno lo marca para ser eliminado. Después los elimina en el momento que considera oportuno.

Justo antes de que un objeto sea eliminado por el recolector de basura, se ejecuta su método finalize(). Si queremos forzar que se ejecute el proceso de finalización de todos los objetos del programa podemos utilizar el método runFinalization() de la clase System. La clase System forma parte de la Biblioteca de Clases de Java y contiene diversas clases para la entrada y salida de información, acceso a variables de entorno del programa y otros métodos de diversa utilidad. Para forzar el proceso de finalización ejecutaríamos:

System.runFinalization();

7 Utilización de métodos.

Los métodos, junto con los atributos, forman parte de la estructura interna de un objeto. Los métodos contienen la declaración de variables locales y las operaciones que se pueden realizar para el objeto, y que son ejecutadas cuando el método es invocado. Se definen en el cuerpo de la clase y posteriormente son instanciados para convertirse en **métodos instancia** de un objeto.

Para utilizar los métodos adecuadamente es conveniente conocer la estructura básica de que disponen.

Al igual que las clases, los métodos están compuestos por una **cabecera** y un **cuerpo**. La cabecera también tiene modificadores, en este caso hemos utilizado public para indicar que el método es público, lo cual quiere decir que le pueden enviar mensajes no sólo los métodos del objeto sino los métodos de cualquier otro objeto externo.

Dentro de un método nos encontramos el cuerpo del método que contiene el código de la acción a realizar. Las acciones que un método puede realizar son:

Inicializar los atributos del objeto
Consultar los valores de los atributos
Modificar los valores de los atributos
Llamar a otros métodos, del mismo del objeto o de objetos externos

7.1 Parámetros y valores devueltos.

Los métodos se pueden utilizar tanto para consultar información sobre el objeto como para modificar su estado. La información consultada del objeto se devuelve a través de lo que se conoce como **valor de retorno**, y la modificación del estado del objeto, o sea, de sus atributos, se hace mediante la **lista de parámetros**.

En general, la lista de parámetros de un método se puede declarar de dos formas diferentes:

Por valor. El valor de los parámetros no se devuelve al finalizar el método, es decir, cualquier modificación que se haga en los parámetros no tendrá efecto una vez se salga del método. Esto es así porque cuando se llama al método desde cualquier parte del programa, dicho método recibe una copia de los argumentos, por tanto cualquier modificación que haga será sobre la copia, no sobre las variables originales.

Por referencia. La modificación en los valores de los parámetros sí tienen efecto tras la finalización del método. Cuando pasamos una variable a un método por referencia lo que estamos haciendo es pasar la dirección del dato en memoria, por tanto cualquier cambio en el dato seguirá modificado una vez que salgamos del método.

En el lenguaje Java, todas las variables se pasan por valor, excepto **los objetos que se pasan por referencia**.

En Java, la declaración de un método tiene dos restricciones:

Un método siempre tiene que devolver un valor (no hay valor por defecto). Este **valor de retorno** es el valor que devuelve el método cuando termina de ejecutarse, al método o programa que lo llamó. Puede ser un tipo primitivo, un tipo referenciado o bien el tipo void, que indica que el método no devuelve ningún valor.

Un método tiene un número fijo de argumentos. Los argumentos son variables a través de las cuales se pasa información al método desde el lugar del que se llame, para que éste pueda utilizar dichos valores durante su ejecución. Los argumentos reciben el nombre de **parámetros** cuando aparecen en la declaración del método.

El valor de retorno es la información que devuelve un método tras su ejecución.
Según hemos visto en el apartado anterior, la cabecera de un método se declara como sigue:

public tipo_de_dato_devuelto nombre_metodo (lista_de_parametros);

Como vemos, el tipo de dato devuelto aparece después del modificador public y se corresponde con el valor de retorno.

La lista de parámetros aparece al final de la cabecera del método, justo después del nombre, encerrados entre signos de paréntesis y separados por comas. Se debe indicar el tipo de dato de cada parámetro así:

(tipo_parámetro1 nombre_parámetro1, ..., tipo_parámetroN nombre_parámetroN)

Cuando se llame al método, se deberá utilizar el nombre del método, seguido de los argumentos que deben coincidir con la lista de parámetros.

7.2 Constructores.

¿Recuerdas cuando hablábamos de la creación e instanciación de un objeto? Decíamos que utilizábamos el operador new seguido del nombre de la clase y una pareja de abrir□cerrar paréntesis. Además, el nombre de la clase era realmente el constructor de la misma, y lo definíamos como un método especial que sirve para inicializar valores. En este apartado vamos a ver un poco más sobre los constructores.

Un constructor es un método especial con el mismo nombre de la clase.

Cuando creamos un objeto debemos instanciarlo utilizando el constructor de la clase. Veamos la clase Date proporcionada por la Biblioteca de Clases de Java. Si queremos instanciar un objeto a partir de la clase Date tan sólo tendremos que utilizar el constructor seguido de una pareja de abrir□cerrar paréntesis:

Date fecha = new Date();

Con la anterior instrucción estamos creando un objeto fecha de tipo Date, que contendrá la fecha y hora actual del sistema.

La estructura de los constructores es similar a la de cualquier método, salvo que no tiene tipo de dato devuelto porque no devuelve ningún valor. Está formada por una cabecera y un cuerpo, que contiene la inicialización de atributos y resto de instrucciones del constructor.

El método constructor tiene las siguientes particularidades:

El constructor es invocado automáticamente en la creación de un objeto, y sólo esa vez.

Los constructores se llaman igual que la clase y las clases.

Puede haber varios constructores para una clase.

Como cualquier método, el constructor puede tener **parámetros** para definir qué valores dar a los atributos del objeto.

El **constructor por defecto** es aquél que no tiene argumentos o parámetros. Cuando creamos un objeto llamando al nombre de la clase sin argumentos, estamos utilizando el constructor por defecto.

Es necesario que toda clase tenga al menos un constructor. Si no definimos constructores para una clase, y sólo en ese caso, el compilador crea un constructor por defecto vacío, que inicializa los atributos a sus valores por defecto, según del tipo que sean: 0 para los tipos numéricos, false para los boolean y null para los tipo carácter y las referencias.

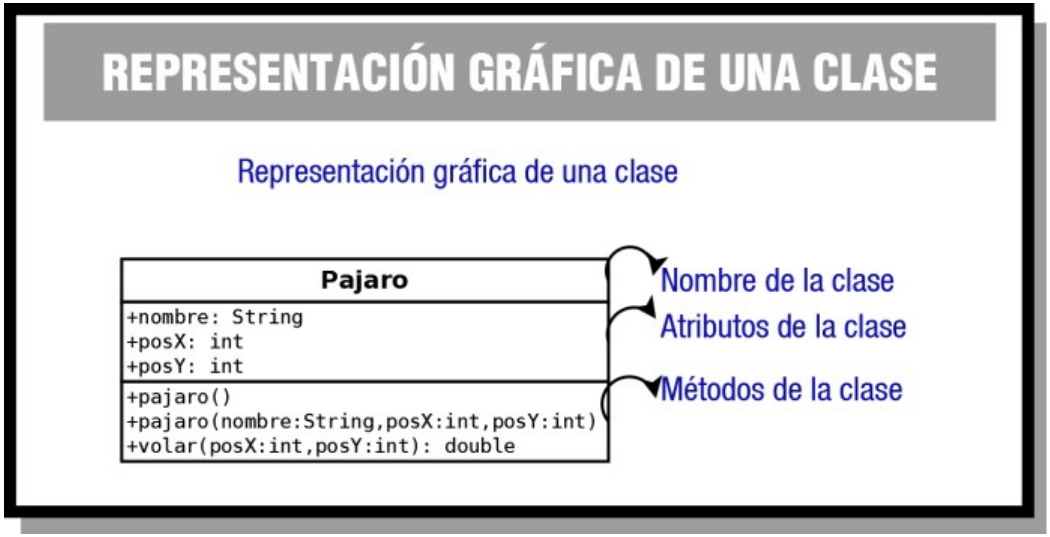
Dicho constructor lo que hace es llamar al constructor sin argumentos de la superclase (clase de la cual hereda); si la superclase no tiene constructor sin argumentos se produce un error de compilación.

Cuando definimos constructores personalizados, el constructor por defecto deja de existir, y si no definimos nosotros un constructor sin argumentos cuando intentemos utilizar el constructor por defecto nos dará un error de compilación.

7.3 El operador this.

Los constructores y métodos de un objeto suelen utilizar el operador this. Este operador sirve para referirse a los atributos de un objeto cuando estamos dentro de él. Sobre todo se utiliza cuando existe ambigüedad entre el nombre de un parámetro y el nombre de un atributo, entonces en lugar del nombre del atributo solamente escribiremos this.nombre_atributo, y así no habrá duda de a qué elemento nos estamos refiriendo.

Vamos a ilustrar mediante un ejemplo la utilización de objetos y métodos, así como el uso de parámetros y el operador this. Aunque la creación de clases la veremos en las siguientes unidades, en este ejercicio creamos una pequeña clase para que podamos instanciar el objeto con el que vamos a trabajar.



Las clases se suelen representar como un rectángulo, y dentro de él se sitúan los atributos y los métodos de dicha clase.

En la imagen, la clase Pajaro está compuesta por tres atributos, uno de ellos el nombre y otros dos que indican la posición del ave, posX y posY. Tiene dos métodos constructores y un método volar().

Como sabemos, los métodos constructores reciben el mismo nombre de la clase, y puede haber varios para una misma clase, dentro de ella se diferencian unos de otros por los parámetros que utilizan.

Ejercicio

Dada una clase principal llamada Pajaro, se definen los atributos y métodos que aparecen en la imagen. Los métodos realizan las siguientes acciones:

pajaro(). Constructor por defecto. En este caso, el constructor por defecto no contiene ninguna instrucción, ya que Java inicializa de forma automática las variables miembro, si no le damos ningún valor.

pajaro(String nombre, int posX, int posY). Constructor que recibe como argumentos una cadena de texto y dos enteros para inicializar el valor de los atributos.

volar(int posX, int posY). Método que recibe como argumentos dos enteros: posX y posY, y devuelve un valor de tipo double como resultado, usando la palabra clave return. El valor devuelto es el resultado de aplicar un desplazamiento de acuerdo con la siguiente fórmula:

$$desplazamiento = \sqrt{posX \cdot posX + posY \cdot posY}$$

7.4 Métodos estáticos.

Los **métodos estáticos** son aquellos métodos definidos para una clase que se pueden usar directamente, sin necesidad de crear un objeto de dicha clase. También se llaman **métodos de clase.**

Para llamar a un método estático utilizaremos:

- El nombre del método, si lo llamamos desde la misma clase en la que está definido.
- El nombre de la clase, seguido por el operador punto (.) más el nombre del método estático , si lo llamamos de una clase distinta a la que está definido.

Los métodos estáticos no afectan al estado de los objetos instanciados de la clase (Variables instancia), y **suelen utilizarse para realizar operaciones comunes a todos los objetos de la clase.** Por ejemplo, si necesitamos contar el número de objetos instanciados de una clase, podríamos utilizar un método estático que fuera incrementando el valor de una variable entera de la clase conforme se van creando los objetos.

En la Biblioteca de Clases String con todas las operaciones que podíamos hacer con ella y con los objetos instanciados a partir de ella. O bien la clase Math. para la conversión de tipos de datos. Todos ellos son métodos estáticos que la API de Java define para esas clases. Lo importante es que tengamos en cuenta que al tratarse de métodos estáticos, para utilizarlos **no necesitamos crear un objeto de dichas clases.**

7.5 Librerías de objetos (paquetes).

Un **paquete** de clases es una agrupación de clases que consideramos que están relacionadas entre sí o tratan de un tema común.

Las clases de un mismo paquete tienen un acceso privilegiado a los atributos y métodos de otras clases de dicho paquete. Es por ello por lo que se considera que los paquetes son también, en cierto modo, unidades de encapsulación y ocultación de información.

Java nos ayuda a organizar las clases en **paquetes**. En cada fichero .java que hagamos, al principio, podemos indicar a qué **paquete** pertenece la clase que hagamos en ese fichero.

Los paquetes se declaran utilizando la palabra clave package seguida del nombre del paquete. Para establecer el paquete al que pertenece una clase hay que poner una sentencia de declaración como la siguiente al principio de la clase:

package Nombre_de_Paquete;

7.6 Sentencia import.

Cuando queremos utilizar una clase que está en un paquete distinto a la clase que estamos utilizando, se suele utilizar la sentencia import. Por ejemplo, si queremos utilizar la clase Scanner que está en el paquete java.util de la Biblioteca de Clases de Java, tendremos que utilizar esta sentencia:

import java.util.Scanner;

Se pueden importar todas las clases de un paquete, así:

import java.awt.*;

Esta sentencia debe aparecer al principio de la clase, justo después de la sentencia package, si ésta existiese.

También podemos utilizar la clase sin sentencia import, en cuyo caso cada vez que queramos usarla debemos indicar su ruta completa:

java.util.Scanner teclado = new java.util.Scanner (System.in);

Hasta aquí todo correcto. Sin embargo, al trabajar con paquetes, Java nos obliga a organizar los directorios, compilar y ejecutar de cierta forma para que todo funcione adecuadamente.

7.7 Compilar y ejecutar clases con paquetes.

Si hacemos que Bienvenida.java pertenezca al paquete ejemplos, debemos crear un subdirectorio "ejemplos" y meter dentro el archivo Bienvenida.java. Tendríamos esta estructura de directorios:

/<directorio_usuario>/Proyecto_Bienvenida/ejemplos/Bienvenida.java

Debemos tener cuidado con las mayúsculas y las minúsculas, para evitar problemas, tenemos que poner el nombre en "package" exactamente igual que el nombre del subdirectorio.

Para **compilar** la clase Bienvenida.java que está en el paquete ejemplos debemos situarnos en el directorio padre del paquete y compilar desde ahí:

$ cd /<directorio_usuario>/Proyecto_Bienvenida

$ javac ejemplos/Bienvenida.java

Si todo va bien, en el directorio ejemplos nos aparecerá la clase compilada Bienvenida.class.

Para ejecutar la clase compilada Bienvenida.class que está en el directorio ejemplos, debemos seguir situados en el directorio padre del paquete. El nombre completo de la clase es "paquete.clase", es decir "ejemplos.Bienvenida". Los pasos serían los siguientes:

$ cd /<directorio_usuario>/Proyecto_Bienvenida

$ java ejemplos/Bienvenida

Si el proyecto lo hemos creado desde un entorno integrado como Netbeans, la compilación y ejecución se realizará al ejecutar la opción RunFile (Ejecutar archivo) o hacer clic sobre el botón Ejecutar.

7.8 Jerarquía de paquetes.

Para organizar mejor las cosas, un **paquete**, en vez de clases, también puede contener otros paquetes. Es decir, podemos hacer **subpaquetes** de los paquetes y subpaquetes de los subpaquetes y así sucesivamente. Esto permite agrupar paquetes relacionados en un paquete más grande. Por ejemplo, si quiero dividir mis clases de ejemplos en ejemplos básicos y ejemplos avanzados, puedo poner más niveles de paquetes separando por puntos:

package ejemplos.basicos;
package ejemplos.avanzados;
A nivel de sistema operativo, tendríamos que crear los subdirectorios básicos y avanzados dentro del directorio ejemplos, y meter ahí las clases que correspondan.

Para compilar, en el directorio del proyecto habría que compilar poniendo todo el path hasta llegar a la clase. Es decir, el nombre de la clase va con todos los paquetes separados por puntos, por ejemplo ejemplos.basicos.Bienvenida.

La estructura de directorios en el sistema operativo cuando usamos subpaquetes sería:

/<directorio_usuario>/Proyecto_Bienvenida/ejemplos/basicos/HolaMundo.java

Y la compilación y ejecución sería:

$ cd /<directorio_usuario>/Proyecto_Bienvenida
$ javac ejemplos/basicos/Bienvenida.java
$ java ejemplos/basicos/Bienvenida

La Biblioteca de Clases de Java se organiza haciendo uso de esta jerarquía de paquetes. Así por ejemplo, si quiero acceder a la clase Date, tendré que importarla indicando su ruta completa, o sea, java.util.Date, así:

import java.util.Date;

7.9 Librerías Java.

Cuando descargamos el entorno de compilación y ejecución de Java, obtenemos la API de Java. Como ya sabemos, se trata de un conjunto de bibliotecas que nos proporciona paquetes de clases útiles para nuestros programas.

Utilizar las clases y métodos de la Biblioteca de Java nos va ayudar a reducir el tiempo de desarrollo considerablemente, por lo que es importante que aprendamos a consultarla y conozcamos las clases más utilizadas.

Los paquetes más importantes que ofrece el lenguaje Java son:

java.io. Contiene las clases que gestionan la entrada y salida, ya sea para manipular ficheros, leer o escribir en pantalla, en memoria, etc. Este paquete contiene por ejemplo la clase BufferedReader que se utiliza para la entrada por teclado.

java.lang. Contiene las clases básicas del lenguaje. Este paquete no es necesario importarlo, ya que es importado automáticamente por el entorno de ejecución. En este paquete se encuentra la clase Object, que sirve como raíz para la jerarquía de clases de Java, o la clase System que ya hemos utilizado en algunos ejemplos y que representa al sistema en el que se está ejecutando la aplicación. También podemos encontrar en este paquete las clases que "envuelven" los tipos primitivos de datos. Lo que proporciona una serie de métodos para cada tipo de dato de utilidad, como por ejemplo las conversiones de datos.

java.util. Biblioteca de clases de utilidad general para el programador. Este paquete contiene por ejemplo la clase Scanner utilizada para la entrada por teclado de diferentes tipos de datos, la clase Date, para el tratamiento de fechas, etc.

java.math. Contiene herramientas para manipulaciones matemáticas.

java.awt. Incluye las clases relacionadas con la construcción de interfaces de usuario, es decir, las que nos permiten construir ventanas, cajas de texto, botones, etc. Algunas de las clases que podemos encontrar en este paquete son Button, TextField, Frame, Label, etc.

java.swing. Contiene otro conjunto de clases para la construcción de interfaces avanzadas de usuario. Los componentes que se engloban dentro de este paquete se denominan componentes Swing, y suponen una alternativa mucho más potente que AWT para construir interfaces de usuario.

java.net. Conjunto de clases para la programación en la red local e Internet.

java.sql. Contiene las clases necesarias para programar en Java el acceso a las bases de datos.

java.security. Biblioteca de clases para implementar mecanismos de seguridad.

Como se puede comprobar Java ofrece una completa jerarquía de clases organizadas a través de paquetes.

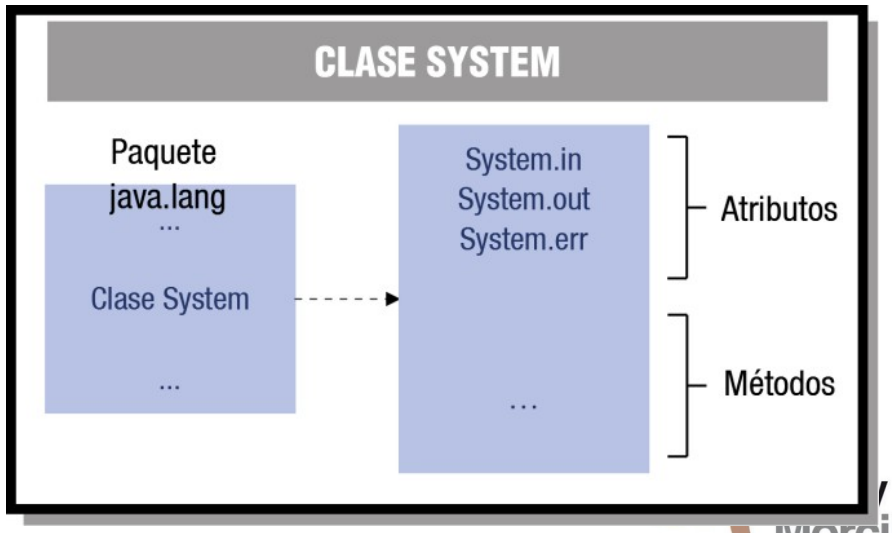
8 Programación de la consola: entrada y salida de la información.

Los programas a veces necesitan acceder a los recursos del sistema, como por ejemplo los dispositivos de entrada/salida estándar, para recoger datos de teclado o mostrar datos por pantalla.

En Java, la entrada por teclado y la salida de información por pantalla se hace mediante la clase System del paquete java.lang de la Biblioteca de Clases de Java.

Como cualquier otra clase, está compuesta de métodos y atributos. Los atributos de la clase System son tres objetos que se utilizan para la entrada y salida estándar. Estos objetos son los siguientes:

System.in. Entrada estándar: teclado.
System.out. Salida estándar: pantalla.
System.err. Salida de error estándar, se produce también por pantalla, pero se implementa como un fichero distinto al anterior para distinguir la salida normal del programa de los mensajes de error. Se utiliza para mostrar mensajes de error.



No se pueden crear objetos a partir de la clase System, sino que se utiliza directamente llamando a cualquiera de sus métodos con el operador de manipulación de objetos, es decir, el operador punto (.):

System.out.println("Bienvenido a Java");

8.1 Conceptos sobre la clase System.

La lectura por teclado es muy importante cuando empezamos a hacer nuestros primeros programas. Para entender mejor en qué consiste la clase System, y en particular el objeto System.in vamos a describirlo más detenidamente.

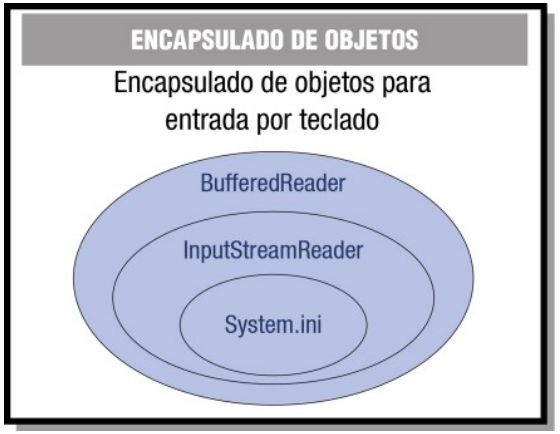
Field Summary

Fields

Modifier and Type	Field and Description
static PrintStream	err The "standard" error output stream.
static InputStream	in The "standard" input stream.
static PrintStream	out The "standard" output stream.

En el apartado anterior hemos dicho que System.in es un atributo de la clase System, que está dentro del paquete java.lang. Pero además, si consultamos la Biblioteca de Clases de Java, nos damos cuenta que es un objeto, y como todos los objetos debe ser instanciado. En efecto, volviendo a consultar la biblioteca de clases nos damos cuenta que System.in es una instancia de una clase de java que se llama InputStream.

En Java, InputStream nos permite leer en bytes, desde teclado, un archivo o cualquier otro dispositivo de entrada. Con esta clase podemos utilizar por ejemplo el método read() que permite leer un byte de la entrada o skip(long n), que salta n bytes de la entrada. Pero lo que realmente nos interesa es poder leer texto o números, no bytes, para hacernos más cómoda la entrada de datos. Para ello se utilizan las clases:



InputStreamReader. Convierte los bytes leídos en caracteres. Particularmente, nos va a servir para convertir el objeto **System.in** en otro tipo de objeto que nos permita leer caracteres.

BufferedReader. Lee hasta un fin de línea. Esta es la clase que nos interesa utilizar, pues tiene un método readLine() que nos va a permitir leer caracteres hasta el final de línea.

La forma de instanciar estas clases para usarlas con System.in es la siguiente:

InputStreamReader isr = new InputStreamReader(System.in);

BufferedReader br = new BufferedReader (isr);

En el código anterior hemos creado un InputStreamReader a partir de System.ini y pasamos dicho InputStreamReader al constructor de BufferedReader. El resultado es que las lecturas que hagamos con el objeto br son en realidad realizadas sobre System.in, pero con la ventaja de que podemos leer una línea completa. Así, por ejemplo, si escribimos una A, con:

String cadena = br.readLine();

Obtendremos en cadena una "A".

Sin embargo, seguimos necesitando hacer la conversión si queremos leer números. Por ejemplo, si escribimos un entero 32, en cadena obtendremos "32". Si recordamos, para convertir cadenas de texto a enteros se utiliza el método estático parseInt() de la clase Integer, con lo cual la lectura la haríamos así:

int numero = Integer.parseInt (br.readLine());

8.2 Entrada por teclado. Clase System.

A continuación vamos a ver un ejemplo de cómo utilizar la clase System para la entrada de datos por teclado en Java.

Como ya hemos visto en unidades anteriores, para compilar y ejecutar el ejemplo puedes utilizar las órdenes javac y java, o bien crear un nuevo proyecto en Netbeans y copiar el código que se proporciona en el archivo siguiente:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
/*
* Ejemplo de entrada por teclado con la clase System
*/
/**
*
* @author FMA
*/
public class EntradaTecladoSystem {
	public static void main(String[] args) {
		try{
			InputStreamReader isr = new InputStreamReader(System.in);
			BufferedReader br = new BufferedReader(isr);
			System.out.print("Introduce el texto: ");
			String cad = br.readLine();
			//salida por pantalla del texto introducido
			System.out.println(cad);
			System.out.print("Introduce un numero: ");
			int num = Integer.parseInt(br.readLine());
			// salida por pantalla del numero introducido
			System.out.println(num);
		} catch (Exception e) {
			// System.out.println("Error al leer datos");
			e.printStackTrace();
		}
	}
}
```

Observa que hemos metido el código entre excepciones try-catch. Cuando en nuestro programa falla algo, por ejemplo la conversión de un String a int, Java nos avisa lanzando excepciones. Si "capturamos" esa excepción en nuestro programa, podemos avisar al usuario de qué ha pasado. Esto es conveniente porque si no tratamos la System excepción seguramente el programa se pare y no siga ejecutándose. El control de excepciones lo veremos en unidades posteriores, ahora sólo nos basta saber que en las llaves del try colocamos el código que puede fallar y en las llaves del catch el tratamiento de la excepción.

8.3 Entrada por teclado. Clase Scanner.

La entrada por teclado que hemos visto en el apartado anterior tiene el inconveniente de que sólo podemos leer de manera fácil tipos de datos String. Si queremos leer otros tipos de datos deberemos convertir la cadena de texto leída en esos tipos de datos.

El kit de Desarrollo de Java, a partir de su versión 1.5, incorpora la clase java.util.Scanner, la cual permite leer tipos de datos String, int, long, etc., a través de la consola de la aplicación. Por ejemplo para leer un tipo de datos entero por teclado sería:

```
Scanner teclado = new Scanner (System.in);

int i = teclado.nextInt ();
```

O bien esta otra instrucción para leer una línea completa, incluido texto, números o lo que sea:

```
String cadena = teclado.nextLine();
```

En las instrucciones anteriores hemos creado un objeto de la clase Scanner llamado teclado utilizando el constructor de la clase, al cual le hemos pasado como parámetro la entrada básica del sistema System.in que por defecto está asociada al teclado.

Para conocer cómo funciona un objeto de la clase Scanner te proporcionamos el siguiente ejemplo:

```
import java.util.Scanner;
/*
* Ejemplo de entrada de teclado con la clase Scanner
*/
/**
*
* @author FMA
*/
public class EntradaTecladoScanner {
      public static void main(String[] args) {
            // Creamos objeto teclado
            Scanner teclado = new Scanner(System.in);
            // Declaramos variables a utilizar
            String nombre;
            int edad;
            boolean estudias;
            float salario;
            // Entrada de datos
            System.out.println("Nombre: ");
            nombre=teclado.nextLine();
            System.out.println("Edad: ");
            edad=teclado.nextInt();
            System.out.println("Estudias: ");
            estudias=teclado.nextBoolean();
            System.out.println("Salario: ");
            salario=teclado.nextFloat();
            // Salida de datos
            System.out.println("Bienvenido: " + nombre);
            System.out.println("Tienes: " + edad +" años");
            System.out.println("Estudias: " + estudias);
            System.out.println("Tu salario es: " + salario +" euros");
      }
}
```

8.4 Salida por pantalla.

La salida por pantalla en Java se hace con el objeto System.out. Este objeto es una instancia de la clase PrintStream del paquete java.lang. Si miramos la API de PrintStream obtendremos la variedad de métodos para mostrar datos por pantalla, algunos de estos son:

void print(String s): Escribe una cadena de texto.
void println(String x): Escribe una cadena de texto y termina la línea.
void printf(String format, Object… args): Escribe una cadena de texto utilizando formato.

En la orden print y println, cuando queramos escribir un mensaje y el valor de una variable debemos utilizar el operador de concatenación de cadenas (+), por ejemplo:

System.out.println(“Bienvenido, ” + nombre);

Escribe el mensaje de "Bienvenido, Carlos", si el valor de la variable nombre es Carlos.

Las órdenes print y println todas las variables que escriben las consideran como cadenas de texto sin formato, por ejemplo, no sería posible indicar que escriba un número decimal con dos cifras decimales o redondear las cifras, o escribir los puntos de los miles, por ejemplo. Para ello se utiliza la orden printf().

La orden printf() utiliza unos códigos de conversión para indicar si el contenido a mostrar de qué tipo es. Estos códigos se caracterizan porque llevan delante el símbolo %, algunos de ellos son:

%c: Escribe un carácter.
%s: Escribe una cadena de texto.
%d: Escribe un entero.
%f: Escribe un número en punto flotante.
%e: Escribe un número en punto flotante en notación científica.

Por ejemplo, si queremos escribir el número float 12345.1684 con el punto de los miles y sólo dos cifras decimales la orden sería:

System.out.printf("% ,.2f\n", 12345.1684);

Esta orden mostraría el número 12.345,17 por pantalla.

Estas órdenes pueden utilizar las secuencias de escape que vimos en unidades anteriores, como "\n" para crear un salto de línea, "\t" para introducir un salto de tabulación en el texto, etc.

8.5 Salida de error.

La salida de error está representada por el objeto System.err. Este objeto es también una instancia de la clase PrintStream, por lo que podemos utilizar los mismos métodos vistos anteriormente.

No parece muy útil utilizar out y err si su destino es la misma pantalla, o al menos en el caso de la consola del sistema donde las dos salidas son representadas con el mismo color y no notamos diferencia alguna. En cambio en la consola de varios entornos integrados de desarrollo como NetBeans o Eclipse la salida de err se ve en un color diferente. Teniendo el siguiente código:

```
System.out.println("Salida estándar por pantalla");
System.err.println("Salida de error por pantalla");
```

La salida de este ejemplo en Netbeans es:

```
run:
Salida estańdar por pantalla
Salida de error por pantalla
BUILD SUCCESSFUL (total time: 1 second)
```

Como vemos en un entorno como Netbeans, utilizar las dos salidas nos puede ayudar a una mejor depuración del código.