Tema 2: Conceptos básicos de Programación Orientada a Objetos

Contenidos

1. Ciclo de vida de un programa
2. ¿Qué es una clase?
3. Composición de clases
4. Herencia de clases
5. ¿Qué es Java?

1 Ciclo de vida de un programa

El ciclo de vida de un programa, también llamado proceso de desarrollo, es el conjunto de etapas en el desarrollo y utilización de una clase.

1. **Especificación**: definición de qué va a hacer un programa.
   Ejemplo: el programa debe representar un cajero automático de un banco en el que se pueda:
   - consultar saldo
   - extraer dinero
   - ingresar dinero

   En esta fase se debe pensar detenidamente en la funcionalidad que se quiere del sistema, en sus casos de uso, etc. En el ejemplo anterior, ¿se permiten operaciones sólo con tarjeta o también con libreta? ¿Se permite operar a clientes de otros bancos o sistemas de tarjetas? etc.
2. **Diseño**: basándose en la información de la fase de especificación, se plantea una solución que determina:
   - qué clases van a hacer falta para representar el problema
   - qué comportamiento (operaciones) van a tener esas clases
   - cómo se relacionan entre ellas

   En esta fase se utilizan muchas veces diagramas (ver Figura 1).
3. **Codificación**: también llamada implementación. Una vez está claro cómo se plantea la solución, hay que codificar las clases. Para ello, hay que basarse en los principios de:

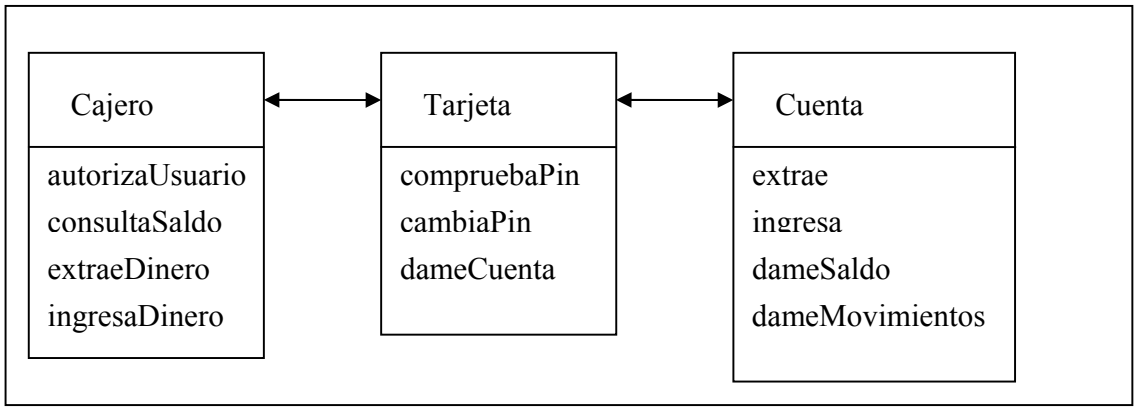

Figura 1: Diagrama de clase fase diseño

- Claridad: el objetivo es que el código sea fácil de entender por el diseñador y por otras personas, para facilitar su depuración, mantenimiento, etc. Para ello, se cuenta con reglas de estilo, de nombrado, de indentación, realización de documentación, etc.
- Eficiencia: se trata de buscar soluciones que, para obtener el mismo resultado, consumar el menor número de recursos posibles. Por recursos se entiende fundamentalmente la memoria y el tiempo de CPU.

4. **Prueba**: se compila y ejecuta el programa para ver si cumple la especificación. Si no lo hace, puede haber errores de codificación o de diseño y hay que volver hacia atrás. ¡Esta fase puede ocupar un 70% del tiempo total de programación! No la descuidéis.
   - Diseñar y codificar las pruebas a la vez que se diseña y codifica el programa
   - Probar el mayor número de casos posibles
   - Probar a la vez que se va codificando, de menor a mayor
     - Cada operación de una clase
     - La clase completa
     - La integración de varias clases
5. **Mantenimiento**: una vez se pone el programa en funcionamiento, no termina su ciclo de vida
   - se pueden detectar nuevos errores
   - se puede querer cambiar o ampliar su funcionalidad

   Por ello es muy importante que el programa sea claro, legible y modular:
   - los cambios no afectarán a todo el programa
   - la persona encargada del mantenimiento probablemente no sea la misma que lo programó

2 ¿Qué es una clase?

La clase es un concepto fundamental de la Programación Orientada a Objetos (POO).

2.1 Definición de clase

Una *clase* agrupa objetos que son idénticos (tienen los mismos atributos y ofrecen las mismas operaciones) salvo en su estado concreto en un momento dado.

- Los atributos representan los datos internos del objeto.
- Las operaciones (llamadas métodos en POO) representan su comportamiento: los mensajes que pueden recibir y procesar de otros objetos.

Ejemplo: vamos a representar mediante objetos a 3 personas, caracterizadas por su nombre, edad, y profesión. Queremos además que los objetos sean capaces de responder a mensajes que pregunten por su nombre, por su edad o por su profesión.

José, 23 años, carpintero
Laura, 37 años, estudiantes
Cristina, 19 años, estudiante

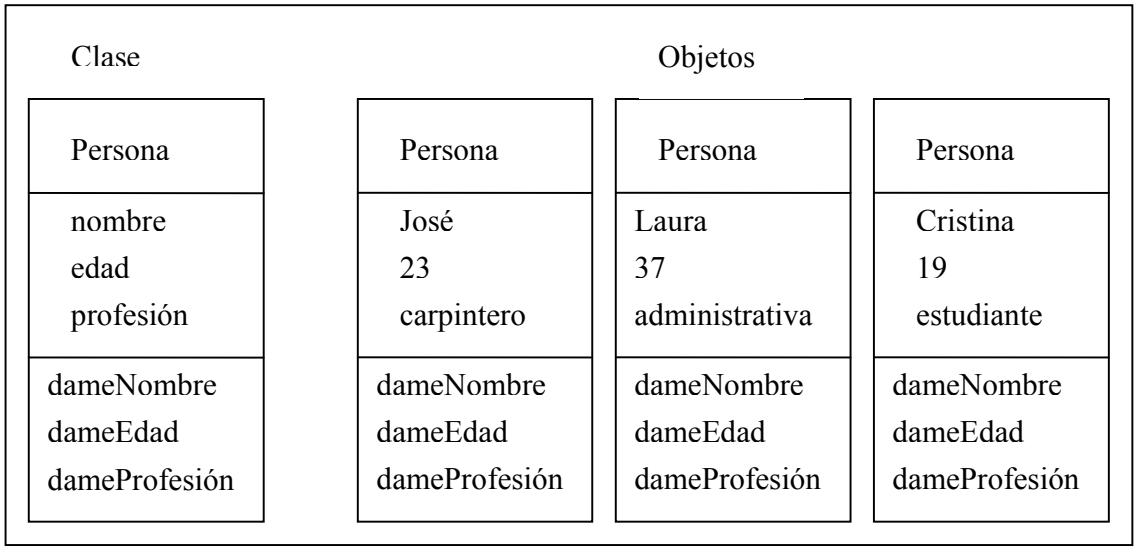

Figura 2: Representación de clases y objetos

Los tres objetos anteriores van a ser instancias de la clase Persona, que se caracteriza por sus atributos (nombre, edad, profesión) y por los métodos que ofrece (dameNombre, dameEdad, dameProfesión), es decir, los mensajes a los que van a ser capaces de responder los objetos de dicha clase.

Más ejemplos: clases para representar circuitos electrónicos, vehículos de motor, asignaturas, etc.

Cada una de las instancias de una clase tiene su propio estado y se puede representar como una entidad única en el programa. A esta entidad única se le llama *objeto*. Cada objeto pertenece a una clase particular que define sus características y su comportamiento. Dada una clase, se pueden crear tantos objetos de esa clase como se quieran.

Una aplicación (un programa) se compone de una serie de clases, de las cuales se crean una serie de instancias (objetos) que interactúan entre sí mandándose mensajes (es decir, llamando a los métodos de los objetos).

2.2 Interfaz e implementación de una clase

- El interfaz de una clase define el tipo de peticiones (operaciones) que se le pueden hacer a objetos de esa clase (ver Figura 3).
  - El interfaz describe qué hace un objeto de una clase
  - A un usuario de una clase sólo le hace falta conocer su interfaz
- La implementación es el código que se programa para satisfacer las peticiones del interfaz.
  - La implementación describe cómo responde un objeto a las peticiones
  - La implementación no necesita ser pública
  - La implementación incluye características internas 8atributos) y el código de las operaciones

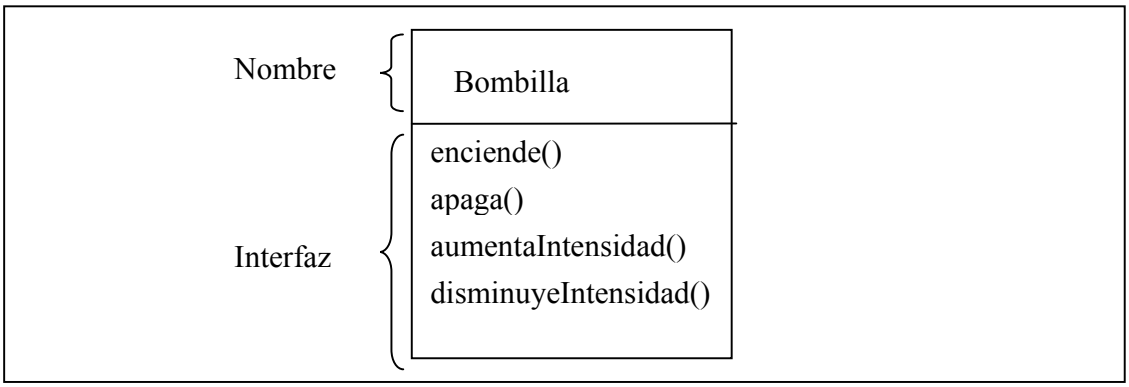

Figura 3: Interfaz de una clase

Los diagramas de clase UML son muy usados para representar gráficamente las clases.

- A veces se omiten los atributos o los atributos y operaciones (métodos), según el nivel de detalle o de visibilidad que se quiera representar en el diagrama

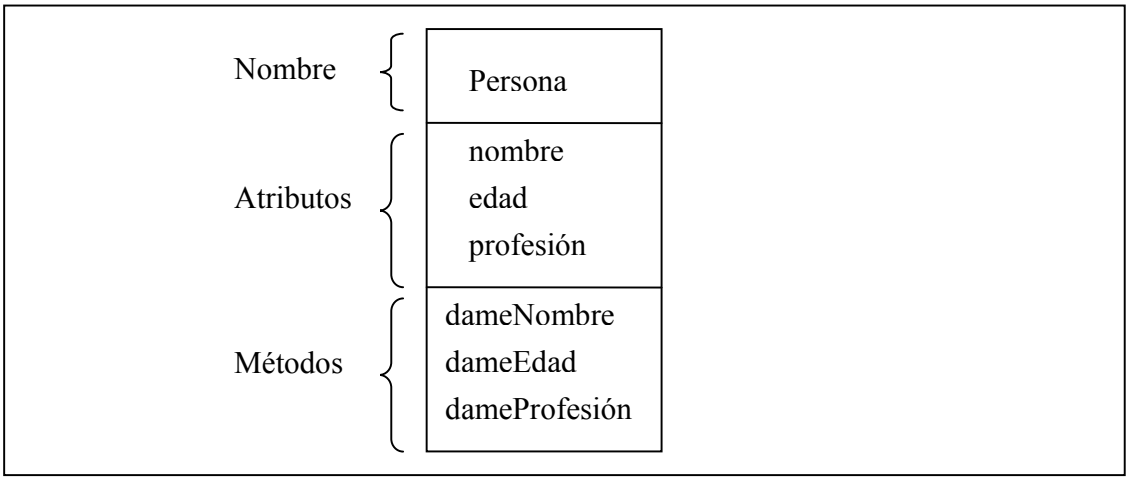

Figura 4: Diagrama de clase UML

3 Composición de clases

La composición es un mecanismo por el que se pueden crear clases complejas agregando objetos de clases ya existentes.

- Se la conoce también con relación "tiene-un"
- Permite reutilizar código

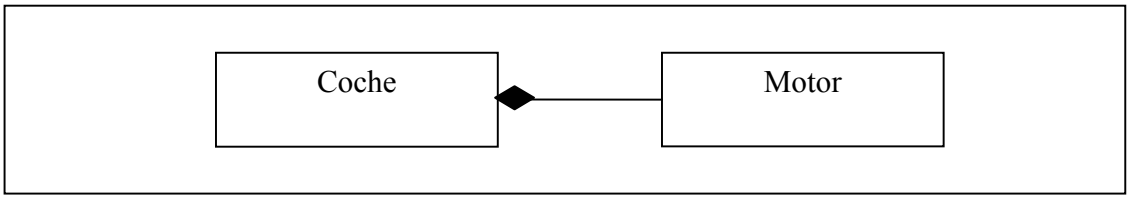

Figura 5: Composición de clases

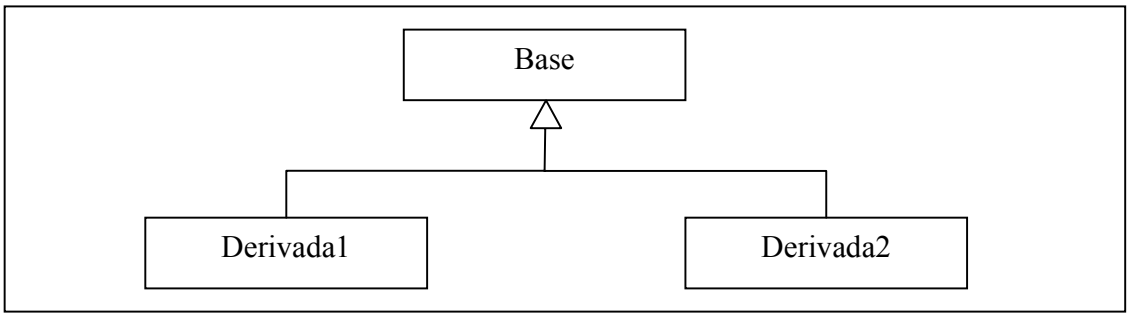

Figura 6: Herencia de clases

La nueva clase (Coche) tiene un objeto miembro (un atributo) que será de la clase Motor.

Más ejemplos de composición de clases: una asignatura tiene un profesor coordinador, una asignatura tiene varios alumnos, un equipo de audio tiene altavoces, etc.

Mediante la composición se permite realizar una programación más modular:

- Los objetos miembros y las clases a las que pertenecen no necesitan ser visibles para clientes de la clase compuesta.
- Aunque cambie la implementación de la clase del objeto miembro, no tengo por qué cambiar la implementación de la clase compuesta (siempre que no cambie la interfaz).

4 Herencia de clases

La herencia es un mecanismo para definir similitud entre clases, en el que se enlazan una clase base y una o varias clases derivadas (ver Figura 6).
La clase Base contiene todas las características (atributos) y comportamientos (métodos) comunes, compartidas por las clases derivadas (el núcleo de las clases "similares")

Las clases Derivadas expresan distintas formas en las que este núcleo se puede realizar (ver ejemplo en Figura 7).

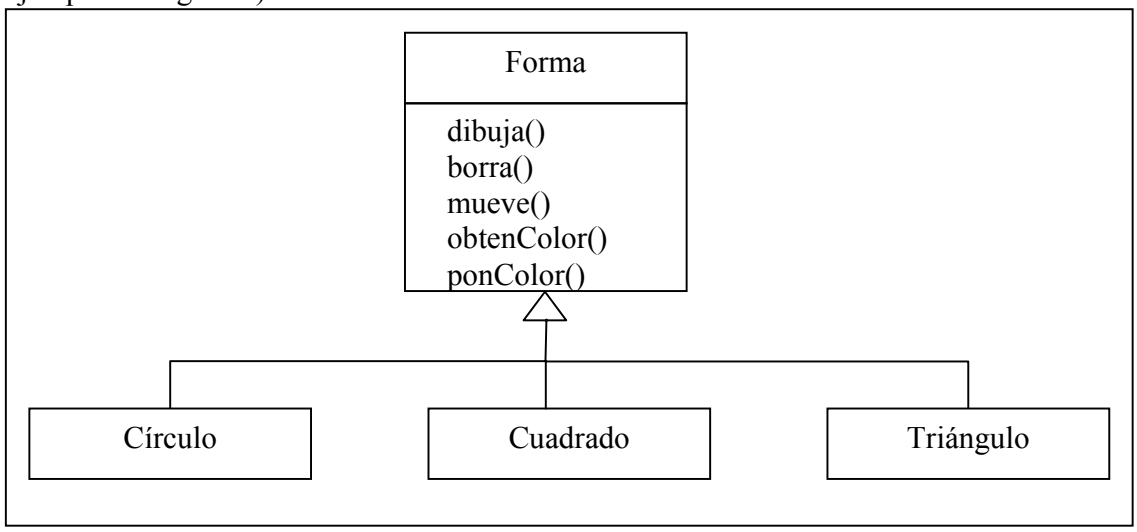

Figura 7: Ejemplo de herencia de clases

Cuando se hereda de una clase Base, se crea una nueva clase que:

- Contiene todos los atributos de la clase Base
- Contiene el interfaz de la clase Base

La herencia se llama también relación "es-un", porque un objeto de la clase Derivada es también un objeto de la clase Base (un Círculo es también una Forma).

4.1 Extensión de clases

La clase Derivada puede añadir nuevos métodos y/o atributos que no serán parte de la clase Base, sino una extensión a la misma

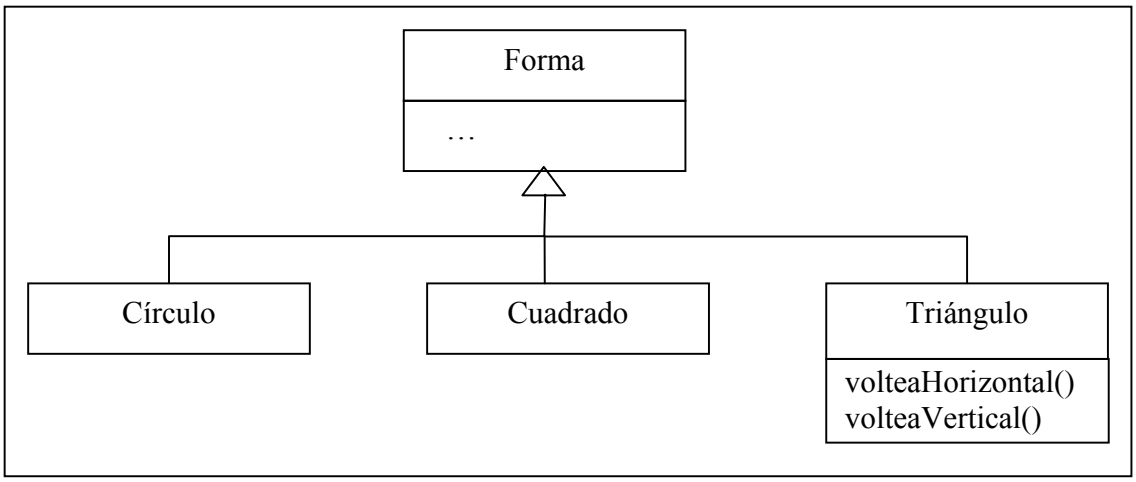

Figura 8: Ejemplo de extensión de clase

4.2 Sobreescritura de métodos

Otra posibilidad de diferenciar entre las clases derivadas es sobrescribir alguno de los métodos. En este caso, las clases derivadas tendrán la misma interfaz pero una implementación distinta.

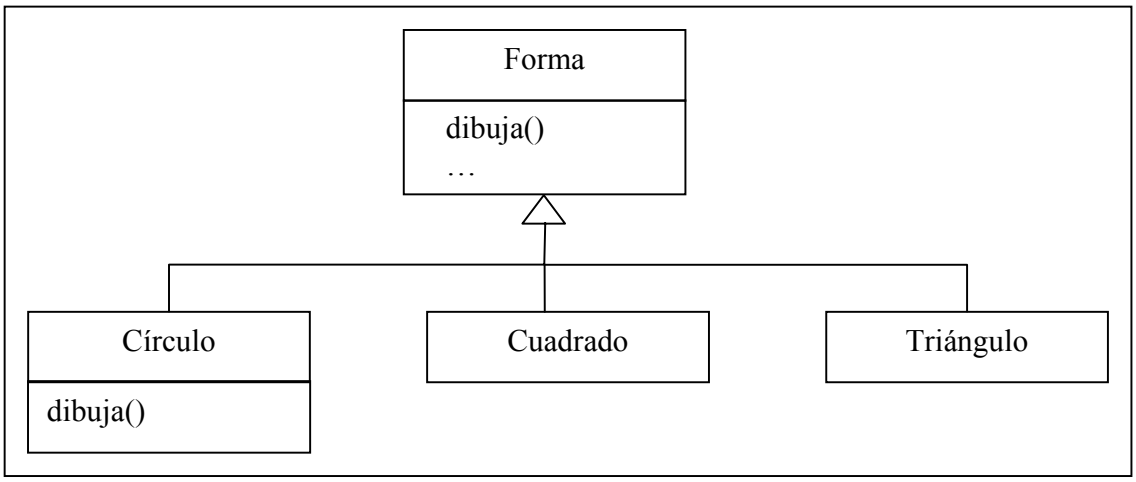

Figura 9: Ejemplo de sobreescritura de método

5 ¿Qué es Java?

Java es un lenguaje de Programación Orientada a Objeto publicado por Sun Microsystems en 1995. Actualmente, la plataforma Java consta de varios elementos:

- El lenguaje de programación Java propiamente dicho
- Un conjunto de bibliotecas estándar que debe existir en cualquier plataforma (Java API)
- Un conjunto de herramientas para el desarrollo de programas:
  - Compilador a bytecode (javac)
  - Generador de documentación (javadoc)
  - Ejecución de programa (intérprete del bytecode (java)
- Un entorno de ejecución cuyo principal elemento es la máquina virtual para ejecutar el bytecode

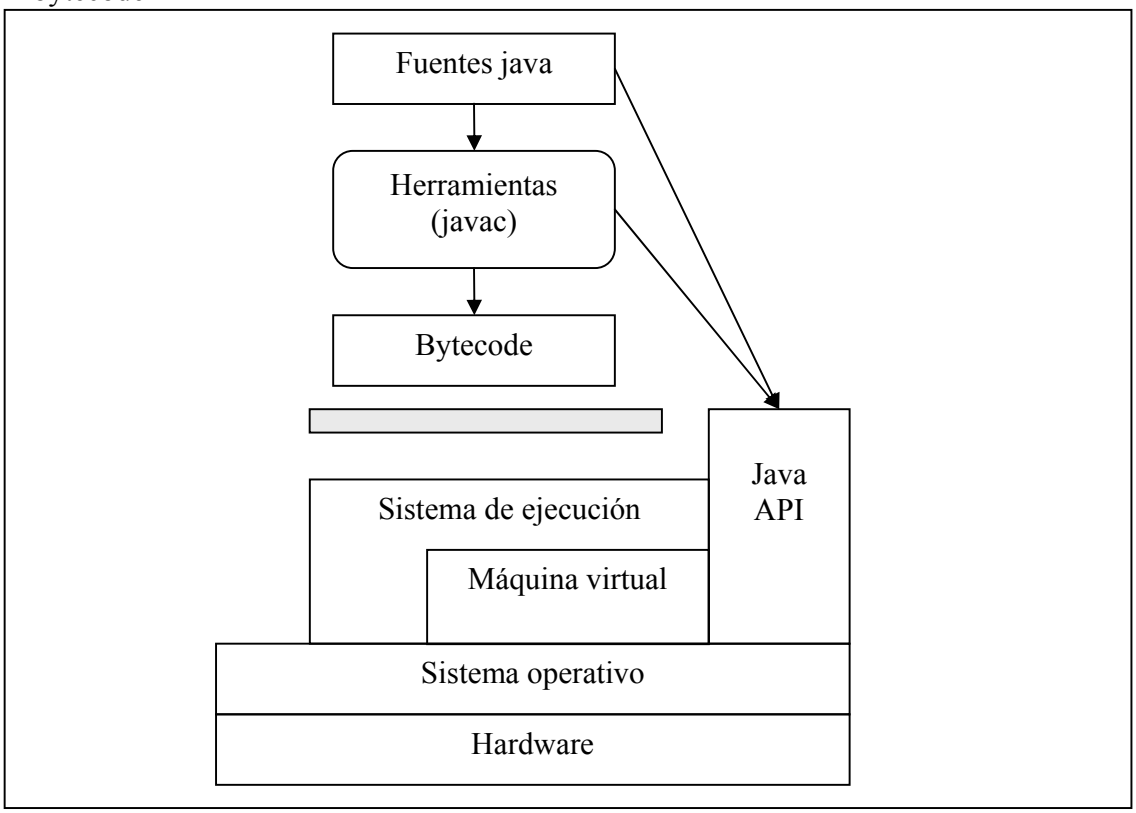

Figura 10: Entorno de ejecución java

5.1 El primer programa en Java

```
/**
* Programa en Java que escribe texto en pantalla
*/
public class Hola {
  public static void main (String[] args){
    System.out.println ("Hola a todos");
    }
}
```

Tema 3: Clases y objetos en Java. Tipos primitivos y operadores

Contenidos

1. Clases y objetos en Java
2. Uso de la memoria
3. Tipos primitivos
4. Elementos sintácticos básicos
5. Operadores
6. Componentes de clase
7. Referencia null y alias

1 Clases y objetos en Java

Para el lenguaje de programación Java aplica la misma definición de clase y objeto que en Programación Orientada a Objetos en general. En Java, "todo es un objeto" – o casi todo, como veremos a lo largo de este tema.

1.1 Definición de clase

En Java, una clase se define mediante la palabra reservada `class`, seguida por el nombre de la clase que vamos definir. El cuerpo de la clase estará rodeado de llaves (`{ }`).

```
class Alumno {
   …
}
```

1.2 Declaración de objetos

Un objeto es una instancia de la clase. Los objetos se declaran indicando en primer lugar el nombre de la clase y después el nombre del objeto:

```
Alumno alumno1;
Alumno alumno2;
```

`alumno1` y `alumno2` son en realidad *referencias* (direcciones de memoria) a objetos de la clase Alumno.

1.3 Creación de objetos

El objeto propiamente dicho se crea con la palabra reservada new y una llamada al constructor. Una vez está creado el objeto, se asigna la dirección de memoria en la que se encuentra a la referencia que se había declarado anteriormente (ver Figura 1).

```
alumno1 = new Alumno();
alumno2 = new Alumno();
```

Ambos pasos (declaración de la referencia, y creación y asignación del objeto) se pueden dar en la misma línea:

```
Alumno alumno1 = new Alumno();
```

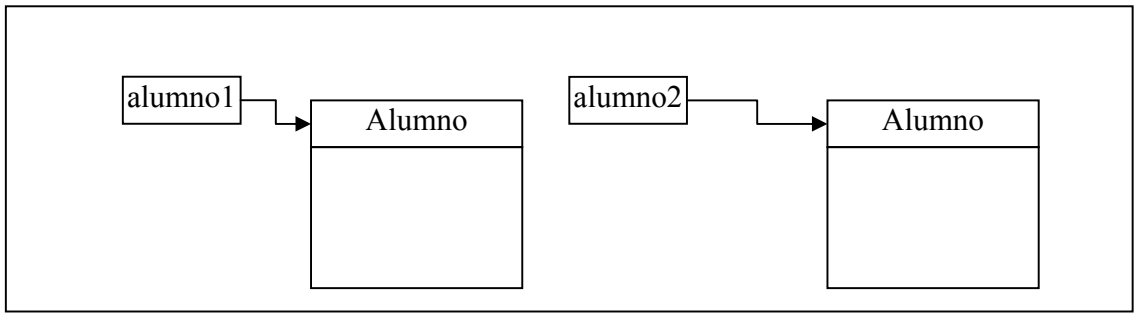

Figura 1: Referencias y objetos

Lo mismo se hace para crear objetos de clases ya definidas en las librerías del lenguaje:

```
Random rand = new Random();
```

2 Uso de la memoria

Una vez sabemos que "todo" en un programa Java son objetos, es interesante saber dónde se colocan estos objetos. En general, es interesante conocer qué tipo de memoria tiene disponible un programa Java y cómo la usa.

1. **Registros**
   Constituyen la memoria de acceso más rápida. Se encuentran dentro del procesador. El número de registros es muy limitado. El acceso a los registros es gestionado directamente por el compilador.
2. **Stack**
   El stack (también llamado pila en español) se encuentra en la RAM.
   Esta zona de la RAM es accedida directamente por el procesador mediante el puntero de stack (*stack pointer*). El puntero de stack:
   - se mueve hacia abajo para obtener nueva memoria
   - se mueve hacia arriba para liberar memoria (ver Figura 2).

   El stack es la segunda memoria más rápida después de los registros.
   En tiempo de compilación, el compilador debe saber el tamaño de todo lo que va en el stack para generar el código que mueva el puntero de stack.
   Dentro del stack se almacenan:
   - instrucciones el programa
   - *datos primitivos*
   - referencias a objetos (¡pero no los propios objetos!)

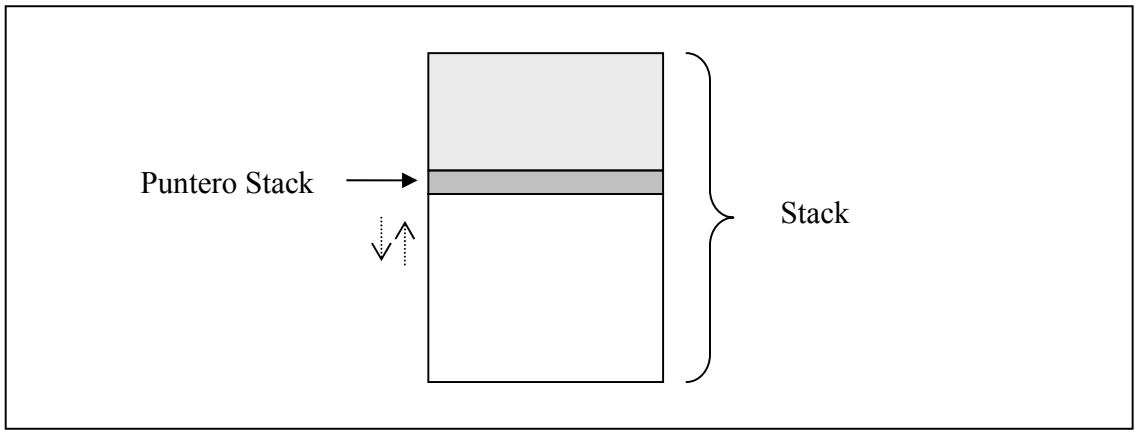

Figura 2: Memoria Stack

3. **Heap**
   El heap es una memoria de propósito general en la RAM donde se ubican los objetos.
   - La reserva se hace en tiempo de ejecución con cada llamada a new.
   - El sistema se ocupa del borrado (garbage collection)
   - Es más lenta de ubicar que la Stack

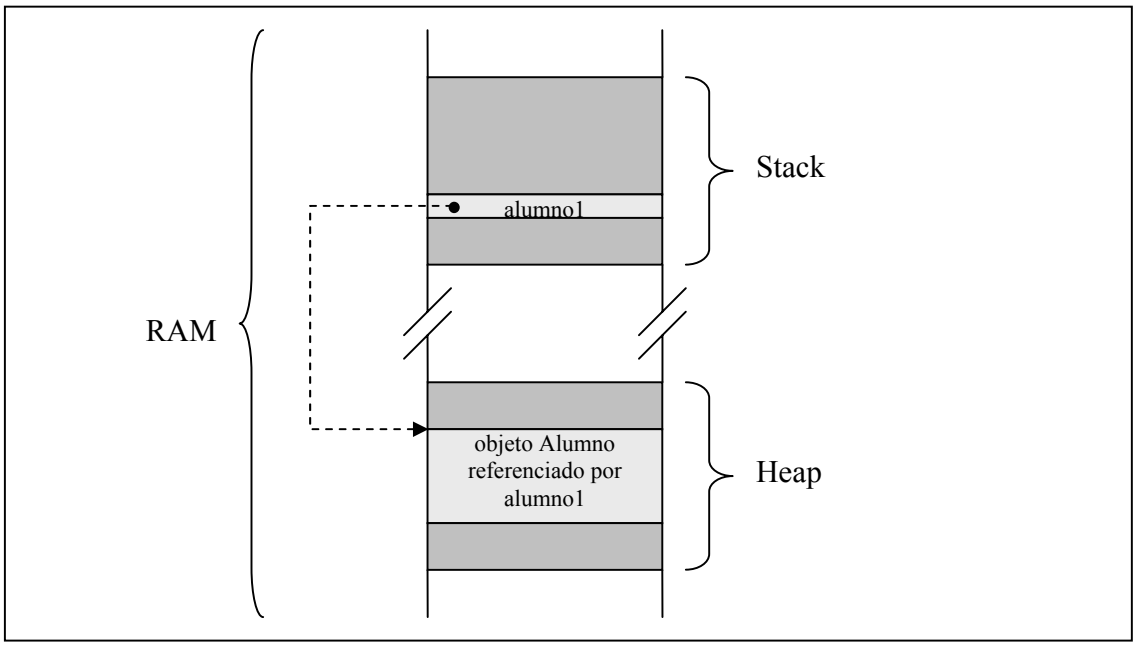

Figura 3: Memoria Stack y Heap

3 Tipos de datos primitivos

Al crear un objeto con `new`, siempre se reserva un espacio en la memoria heap, aunque se trate de variables pequeñas y sencillas. Esto no es muy eficiente.

Java permite usar variables de tipos primitivos.

- no es POO pura (no todo son objetos)

- las variables no son referencias, es decir, no apuntan a posiciones de memoria, sino que contienen un valor
- las variables son de tamaño fijo, también entre distintas máquinas, por lo que el código es portable
- las variables se almacenan en el stack, de acceso más rápido

Los tipos primitivos definidos en Java se muestran en la Tabla 1.

Tabla 1: Tipos de datos primitivos en Java

Tipo primitivo	Descripción	Tamaño	Mínimo	Máximo	Envoltorio
boolean	Valor binario	-	-	-	Boolean
char	Carácter unicode	16 bits	unicode 0	unicode 2^{16}-1	Character
byte	Entero con signo	8 bits	-128	+127	Byte
short	Entero con signo	16 bits	-2^{15}	$+2^{15}$-1	Short
int	Entero con signo	32 bits	-2^{31}	$+2^{31}$-1	Integer
long	Entero con signo	64 bits	-2^{63}	$+2^{63}$-1	Long
float	Real simple precisión	32 bits	$\pm 3.40282347\ e^{+38}$ $\pm 1.40239486\ e^{-45}$		Float
double	Real doble precisión	64 bits	$\pm 1.79769313486231570\ e^{+308}$ $\pm 4.94065645841246544\ e^{-324}$		Double
void	- vacío -	-	-	-	Void

Las **clases envoltorio** (*wrapper*) permiten tratar una variable primitiva como un objeto:

```
char c = 'x';
Character c1 = new Character (c);
Character c2 = new Character ('x');
```

4 Elementos sintácticos básicos

A continuación se va a hacer una revisión de los elementos sintácticos básicos de Java por medio de un ejemplo.

```
/** Programa Java que dado el radio y altura
 *  de un cilindro calcula su volumen
 */
public class VolumenCilindro {
  public static void main (String[] args){
    /* El valor del número pi */
    final double PI = 3.1415926536;
    double radio;
    double altura;

    radio = 23.4;   // en centímetros
    altura = 120.2; // en centímetros
    System.out.println("Datos del cilindro: ");
    System.out.println("Radio: " + radio);
    System.out.println("Altura: " + altura);
    System.out.println("El volumen del cilindro es: ");
    System.out.println(PI * radio * radio * altura);
  }
}
```

4.1 Comentarios

- `/* ... */`: comentario de bloque. Todo lo que aparece entre las dos marcas se considera un comentario
- `/** ... */`: comentario de documentación. Es un caso particular de los comentarios de bloque. Permite la inclusión del comentario en la documentación del programa generada automáticamente con la herramienta `javadoc`
- `//`: comentario de línea. Desde la marca al final de esa línea se considera un comentario

4.2 Palabras reservadas

Son palabras que forman parte del lenguaje Java propiamente dicho y no pueden usarse como nombres de clases, objetos, etc. Ejemplos de palabras reservadas: `class`, `new`, `int`, `void`, `final`, etc.

4.3 Identificadores

Son nombres que elige el programador para distintos elementos del programa: clases, objetos, variables, métodos, etc. Deben empezar con una letra y seguir con una sucesión de letras y cifras.

En este sentido se consideran letras válidas para formar identificadores:

- Caracteres UNICODE: alfabeto latino, griego, hebreo, cirílico, con acentos, etc
- Los siguientes caracteres: subrayado (_) y dólar (`$`)

Ejemplos de identificadores: `VolumenCilindro`, `PI`, `radio`, `altura`

Convenios de nombrado de identificadores:

- Identificadores de variables o métodos: empezar con minúscula. Si es un nombre compuesto, empezar cada palabra posterior con mayúscula. Ejemplo: `númeroElementos`, `ponValor`
- Identificadores de clases: mismas consideraciones que en el caso anterior, pero empezando por mayúscula. Ejemplo: `VolumenCilindro`
- Identificadores de constantes: se escriben todo en mayúsculas. Si es un nombre compuesto, se separa cada palabra de la posterior con un subrayado. Ejemplo: `PI`, `VALOR_MAXIMO`

4.4 Variables

Son nombres que hacen referencia a:

- Objetos de una clase
- Elementos de tipos primitivos

Una variable queda determinada por:

- Un nombre
- La clase a la que pertenece el objeto o bien el tipo primitivo (que determina qué valores se pueden guardar y con qué rango)

Declaración de variables

Se escribe primero el nombre del tipo o clase, y luego el nombre de la variable

```
double radio;
```

Asignación de variables
Se escribe el nombre de variable seguido de un signo de igual seguido de una expresión del mismo tipo o clase que la variable a la que se le está asignando un valor.

```
radio = 23.4;
radio = 12.4 + 4.2 + a;
```

La declaración y asignación de variables pueden realzarse a la vez:

```
double radio = 23.4;
String texto = "Representación de Datos y aplicaciones";
```

Constantes
Algunas "variables" se declaran como constantes. Eso quiere decir que su valor no puede cambiar en toda la ejecución del programa. Se preceden por la palabra reservada `final`.

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

Inicialización de variables
Java inicializa todas las variables de tipo primitivo que son atributos de la clase a un valor por defecto que es cero para los tipos numéricos, el carácter UNICODE con código 0 para los caracteres y `false` para los `boolean`.

Esto sólo sucede para variables que sean atributos de clase:

```
class SoloDatos {
  int i;
  float f;
  boolean b;
}
```

Pero no para variables locales, definidas por ejemplo dentro de un método:

```
void metodoEjemplo () {
  int i;
  ...
}
```

En este caso no se garantiza un valor inicial de la variable, por lo que conviene inicializarla antes de usarla.

4.5 Literales

Se llama literales a posibles valores de tipos primitivos predefinidos en el lenguaje.

Números enteros
Por defecto (si no se especifica otra cosa) se suponen del tipo `int`.

- se interpreta como `long` si se añade el carácter '`l`' o '`L`' al final: `21L`

Pueden representarse de 3 formas:

- Decimal. Ejemplo: 21
- Octal (base 8: cifras del 0-7). Se añade un 0 por delante: 025
- Hexadecimal (base 16: caracteres del 0-9,A-F). Se añade un 0x delante: 0x15

Números reales

Por defecto (si no se especifica otra cosa) se suponen del tipo `double`.

- se interpreta como `float` si se añade el carácter '`f`' o '`F`' al final: `2.31f`

Pueden representarse de 2 formas:

- Con un punto (ojo, no coma) decimal. Ejemplos: 2.1, 2., .54
- Con notación exponencial. Ejemplo: 1e2 = 1 * 10^2

Booleanos

Pueden tomar los valores `true` o `false`, en minúscula.

Caracteres

Pueden tomar cualquier valor de la tabla de caracteres UNICODE. Hay dos tipos de representación:

- carácter entre comillas simples. Ejemplo: 'a', 'Ñ'
- código UNICODE en hexadecimal ('\u00A3') u octal ('\102')

Existe una serie de caracteres especiales:

- `\b` : retroceso
- `\t` : tabulador
- `\n` : salto de línea
- `\r` : cambio de línea
- `\"` : carácter comillas dobles
- `\'` : carácter comillas simples
- `\\` : carácter barra atrás

Textos (String)

String no es un tipo de datos primitivo sino una clase definida en Java para representar cadenas de textos. Es decir, las variables String son referencias al objeto de tipo String.

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

Las cadenas de texto son una de las variables más utilizadas en Java, por eso hay una forma especial abreviada de crearlas:

```
String s = "asdf";
```

Esta línea es equivalente a la anterior. El contenido de la cadena de texto se escribe entre comillas dobles. Ejemplos: "perro", " " (cadena con un espacio en blanco), "" (cadena vacía), "a" (cadena formada por el carácter 'a': "a" ≠ 'a')

5 Operadores

Un operador toma uno o más argumentos y produce un nuevo valor.

Casi todos los operadores trabajan con tipos de datos primitivos, excepto:

- == y `!=`, operadores de comparación, que trabajan con todo tipo de objetos
- + y += para concatenación de Strings

Precedencia
La precedencia de los operadores define cómo se evalúa una expresión cuando hay varios operadores presentes. Por ejemplo:

```
a = x + y - 2 / 2 + z
```

es distinto de

```
a = x + (y - 2) / (2 + z)
```

El orden de precedencia de los operadores más comunes es:

1. Operadores unarios
2. Operadores multiplicativos (de izquierda a derecha)
3. Operadores aditivos (de izquierda a derecha)
4. Operadores de relación
5. Operadores lógicos
6. Operadores de asignación

En caso de duda, usad paréntesis.

5.1 Operadores aritméticos

Son operadores binarios (necesitan dos argumentos). Realizan las operaciones clásicas de la aritmética:

`(+)` : adición
`(-)` : sustracción
`(*)` : multiplicación
`(/)` : división
`(%)` : módulo (resto)

Advertencias:

- la división entera trunca y no redondea
- el resultado de operaciones con números reales es un número real

Operación y asignación
Existen abreviaturas para realizar a la vez la operación y la asignación. En estos casos, la variable que representa a uno de los operandos es la misma sobre la que se escribe el resultado: (+=), (-=), (*=), (/=), (%=)

`a += 4` es equivalente a `a = a + 4`

5.2 Operaciones unarias

Operaciones de signo

El operador (-) unario produce un cambio de signo. El operador unario (+) no produce ningún efecto, existe por simetría con el anterior. Ejemplos:

```
a = -b;
x = a * -b;
```

Operaciones de autoincremento y autodecremento

(++) y (--): aumentan o decrementan el valor de la variable en uno. Existen dos variantes:

- Preincremento (Predecremento): ++a (--a): primero se incrementa (decrementa) y luego se produce el valor.
- Postincremento (Postdecremento): a++ (a--): primero se produce el valor y luego se incrementa (decrementa).

Ejemplo:

```
int i = 1;
System.out.println(i);
System.out.println(++i);
System.out.println(i);
System.out.println(i++);
System.out.println(i);
```

produciría en pantalla:

```
1
2
2
2
3
```

5.3 Operadores relacionales

Los operadores relacionales producen un resultado booleano:

- `true`: si la relación es cierta
- `false`: si la relación es falsa

Tipos de operadores relacionales:

- (==), `(!=)`: operan con todos los objetos
  - no usar comparadores de igualdad/desigualdad con números reales (precisión finita)
  - en el caso de variables de tipo primitivo, comparan los valores
  - en el caso de referencia, comparan los valores… de las referencias (no del contenido de los objetos. Hay que tener cuidado con los String porque son referencias a objetos. Para comparar los objetos predefinidos del lenguaje, usar `equals`

    ```
    Integer n1 = new Integer (47);
    Integer n2 = new Integer (47);
    ```

```
System.out.println (n1 == n2);        //false
System.out.println (n1.equals(n2));   //true
```

- (>), (<), (>=), (<=): funcionan con todos los tipos primitivos menos con boolean

5.4 Operadores lógicos

Tanto los operandos como el resultado son booleanos.

- (`&&`): AND lógico
- (`||`): OR lógico
- (`!`): NOT lógico

Ejemplo: vamos a calcular la expresión lógica de la implicación: a => b. La tabla de verdad de la implicación lógica es:

a	b	=>
0	0	1
0	1	1
1	0	0
1	1	1

La expresión lógica equivalente sería: `! (a && !b)`
El orden de precedencia de los operadores lógicos es:

1. negación
2. and
3. or

Expresiones aritmético-lógicas
Estas expresiones constituyen una mezcla de expresiones aritméticas, relacionales y lógicas. Por ejemplo:

```
3+7 < 4*3 || 9 < 3
```

Efecto cortocircuito
Los operadores lógicos (&&) y (||) evalúan primero el operando de su izquierda y luego deciden si necesitan continuar evaluando el otro operador, teniendo en cuenta que:

```
false && a es false
true || a es true
```

Por ejemplo, esta expresión no tendría problemas:

```
(y != 0) && (x/y > 1)
```

5.5 Operadores sobre Strings

El operador más importante sobre Strings es el operador de concatenación (+). Usa el mismo símbolo que la suma. Por ejemplo:

```
int i = 7;
```

```
System.out.println("Resultado: " + i);
```

El valor del entero `i` se convierte automáticamente en una cadena de caracteres para poder ser concatenado con el String anterior.

5.6 Conversión de tipo

- Conversión ascendente: un operando en una expresión híbrida se convierte automáticamente al tipo mayor (con mayor número de valores). Ejemplo:

```
double gradosC, gradosF;
gradosF = 90.2;
gradosC = (gradosF – 32) * 5 / 9;
```

- Conversión ascendente: la conversión de un valor de un tipo a otro de un tipo menor (con menor número de valores) se tiene que realizar explícitamente. Ejemplos:

```
gradosEnteros = (int) ((gradosF – 32) * 5 / 9);

char c = 'A';
c = (char) (c+2);  // c == 'C'
```

6 Componentes de clase

Una vez se han presentado los tipor primitivos y los operadores básicos, vamos a analizar en detalle los componentes de una clase: atributos y métodos.

6.1 Atributos

Los atributos son elementos que definen el estado de un objeto. Los atributos pueden ser variables de tipo primitivo o referencias a objetos. Se puede acceder a los atributos de un objeto mediante el nombre del objeto, seguido de un punto, seguido del nombre del atributo. Ejemplo:

```
class Alumno {
  String nombre;
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal;    //identificativo único
  String grupo;
  char horario;          //'M': mañana, 'T': tarde
  ...
}

Alumno alumno1 = new Alumno();
alumno1.nombre = "Pedro";
```

6.2 Métodos

También llamados operaciones o funciones miembro. Los métodos determinan los mensajes que puede recibir un objeto, es decir, definen el comportamiento del objeto. La forma de definir un método es:

```
<visibilidad> <tipoRetorno> <nombreMétodo> ( <listaArgumentos>) {
  <cuerpoMétodo>
}
```

Ejemplo:

```
class Alumno {
  ...
  public String dameGrupo () { ... }
  public void ponGrupo(String nuevoGripo) { ... }
  ...
}

Alumno alumno1 = new Alumno();
alumno1.ponGrupo("7031-91");
System.out.println("El grupo de " + alumno1.nombre + " " +
        alumno1.apellidos + " es el " + alumno1.dameGrupo());
```

7 Referencia `null` y alias

Referencia null

Una referencia a un objeto puede no tener asignada ninguna instancia. Esto se representa asociando a esa referencia el valor especial `null` (válido para todas las clases de objetos).

Ejemplo:

```
Alumno alumno1;               //valen null por defecto
Alumno alumno2;
Alumno alumno3;

alumno1 = new Alumno();       // vale /= null
alumno2 = new Alumno();       // vale /= null
alumno2 = null;               // vale null por asignación
```

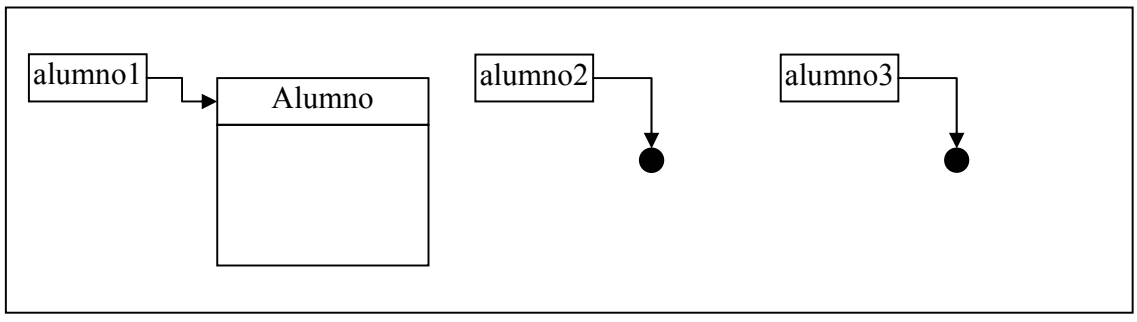

Figura 4: Referencia a null

Es importante tener en cuenta que no se puede acceder a ningún atributo o método de una referencia que vale null porque no existe ningçun objeto asociado. Daría como resultado la excepción (error) NullPointerException.

```
alumno3.ponGrupo("7031-91");       // error!!
```

Por lo tanto, antes de usar una referencia hay que comprobar que no sea null.

```
if (alumno3 != null)
   alumno3.ponGrupo("7031-91");       // correcto
```

Se ha usado la sentencia condicional if, que se verá en detalle en temas posteriores. La forma general de esta sentencia es

```
if (condición) {
  sentencias
} else {
  sentencias
}
```

Alias

Un objeto puede tener varias referencias, que se conocen como alias. Por ejemplo:

```
Alumno delegado;
delegado = alumno1;
```

Lo que se está copiando es la referencia y no el objeto. Imaginemos un escenario como el mostrado en la Figura 5. ¿Qué resultados darían las comparaciones de las distintas referencias?

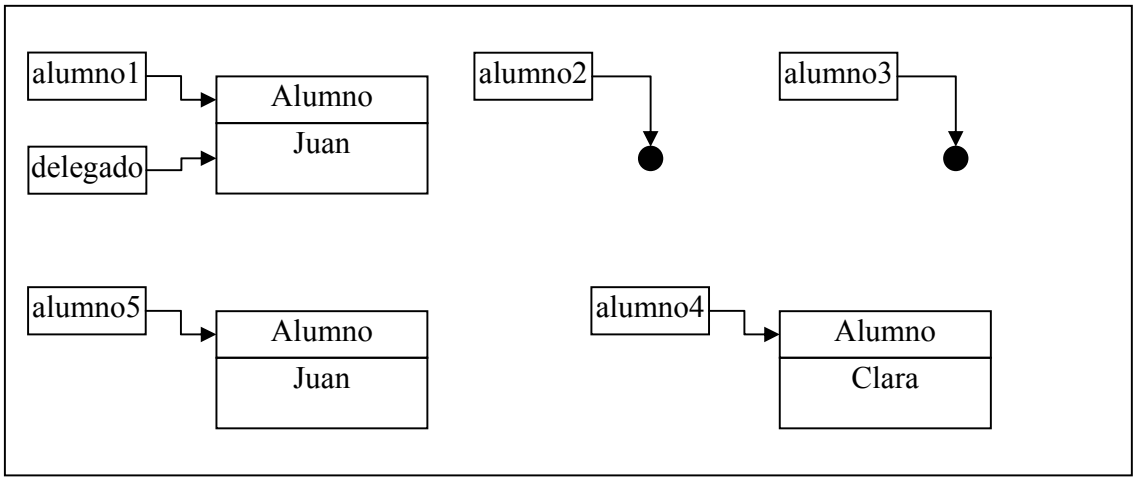

Figura 5: Alias de referencias

Tema 4: Clases: atributos y métodos

Contenidos

1. Atributos
2. Métodos
3. Constructores

1 Atributos

Los atributos son elementos que definen el estado de un objeto. Los atributos pueden ser variables de tipo primitivo o referencias a objetos.

```
class Alumno {
  String nombre;
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal
  String grupo;
  char horario
}
```

- Los atributos pueden inicializarse en el momento de declararse. Por ejemplo:

```
class Alumno {
  String nombre;
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal
  String grupo;
  char horario = 'M';
}
```

- Los atributos que no se inicialicen tienen un valor por defecto, que será 0 para números y caracteres, y null para referencias
- Los atributos suelen inicializarse en el constructor, un método especial que veremos más adelante
- Los atributos de un objeto se pueden acceder de varias formas:

1. Con el operador punto (.):
   Por ejemplo:

```
class Alumno {
  String nombre;
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal
  String grupo;
  char horario

  public static void main (String [] args) {
    Alumno alumno1 = new Alumno();
    alumno1.nombre = "Pedro";
```

```
      alumno1.añoDeNacimiento = 1985
    }
}
```

2. Directamente con el nombre del atributo, si accedemos a él desde un método no estático de la clase (o sea, desde un método que no sea el main). Esto es así porque cuando llamamos a un método es siempre sobre un objeto concreto
   Por ejemplo:

```
class Alumno {
  String nombre;
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal
  String grupo;
  char horario

  String dameGrupo() {
    return grupo;
  }
}
```

3. Si accedemos a él desde un método no estático de la clase, como en el caso anterior, y puede haber confusión en el nombre, entonces se puede acceder de la forma `this.atributo`. El `this` se llama también autoreferencia, y representa siempre al objeto sobre el que se está accediendo. El valor lo va asignando automáticamente el sistema de ejecución Java.
   Por ejemplo:

```
class Alumno {
  String nombre;
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal
  String grupo;
  char horario

  void ponGrupo (String grupo) {
    this.grupo = grupo;
  }
}
```

2 Métodos

También llamados operaciones o funciones miembro. Los métodos determinan los mensajes que puede recibir un objeto, es decir, definen el comportamiento del objeto. La forma de definir un método es:

```
<visibilidad> <tipoRetorno> <nombreMétodo> ( <listaArgumentos>) {
  <cuerpoMétodo>
}
```

Por ejemplo:

```
class Alumno {
  String nombre;
```

```
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal
  String grupo;
  char horario = 'M';

  public String dameGrupo () {
    return grupo;
  }
  public void ponGrupo(String nuevoGrupo) {
    grupo = nuevoGrupo;}

  // otras definiciones de métodos
}
```

Vamos a explicar ahora los elementos de la definición de un método:

- *Visibilidad*. Determina desde qué partes del programa se va a poder acceder al método. Este concepto se explicará más adelante en detalle. Un ejemplo de identificador de visibilidad es `public`.
- *Tipo de retorno*. Es el tipo primitivo o clase del resultado del método. El resultado del método se define en la sentencia `return`.
- *Lista de parámetros*. Da el tipo y nombre de 0, 1 o varios parámetros pasados al método. Si son varios, van separados por comas.
- *Cuerpo del método*. Es la parte en la que se indica qué acciones se realizan al llamar a dicho método, es decir, define el comportamiento del método

2.1 Invocación del método

Cuando tenemos un objeto de cierta clase, podemos invocar sobre él cualquiera de los métodos definidos en la clase. La forma de invocar un método es similar a la forma de invocar atributos.

1. Si se utiliza desde fuera de la clase, se pone la referencia al objeto, el operador punto, el nombre del método, y los valores concretos para los parámetros del método entre paréntesis. Por ejemplo:

```
class Alumno {
  String nombre;
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal
  String grupo;
  char horario = 'M';

  public String dameGrupo () {
    return grupo;
  }
  public void ponGrupo(String nuevoGrupo) {
    grupo = nuevoGrupo;
  }
```

```
  public static void main (String[] args) {
    Alumno alumno1 = new Alumno();
    alumno1.ponNombre("Luis");
    String nombreAlumno = alumno1.dameNombre();
  }
}
```

2. Directamente con el nombre del método, si accedemos a él desde un método no estático de la clase (o sea, desde un método que no sea el main). Esto es así porque cuando llamamos a un método es siempre sobre un objeto concreto. Por ejemplo:

```
class Alumno {
  String nombre;
  String apellidos;
  int añoDeNacimiento;
  int númeroPersonal
  String grupo;
  char horario = 'M';

  public void ponGrupo(String nuevoGrupo) {
    grupo = nuevoGrupo;
  }
  public void ponGrupoHorario(String grupo, char horario) {
    ponGrupo(grupo);
    this.horario = horario;
  }
}
```

3. Si es un método de clase (un método estático definido con la palabra `static`) se pone el nombre de la clase, el punto, el nombre del método y la lista de valores para los parámetros entre paréntesis. Por ejemplo:

```
int numero = Integer.parseInt("37");
```

2.2 Resultado de la llamada al método

El resultado de la llamada a un método se define en la sentencia `return`. Tras ejecutarse la sentencia `return`, el método termina. Hay tres formas básicas de terminar un método y das el resultado:

1. Escribiendo la palabra reservada `return` seguida del valor resultante. Por ejemplo, `return grupo;`.
2. Hay métodos que no devuelven resultados (los que declaran su tipo resultado como `void`). En esos casos, se utiliza la palabra sentencia `return;`, sin indicar el resultado. La ejecución del método termina en este momento
3. Si el método no devuelve ningún valor, es posible suprimir la sentencia `return` (como en el ejemplo de `ponGrupo`). En ese caso, la ejecución del método termina cuando se llega al final del cuerpo del método (`}`).

El valor resultado de la llamada al método puede usarse directamente en el programa llamante. El tipo del resultado tiene que ser compatible con el tipo de la variable a la que se asigna:

```
String grupoAlumno1 = alumno1.dameGrupo();
```

2.3 Parámetros del método

Se llaman parámetros a los datos que un método necesita para su ejecución y que se detallan en la cabecera del método.
Se llaman argumentos a los valores concretos que se pasan para esos parámetros en la llamada al método.
En la llamada, se puede poner como argumento un valor literal del tipo, o una variable, o una expresión, o el resultado de una llamada a función:

```
alumno1.ponNombre("Pedro");
alumno1.ponNombre(nombreAlumno1);
alumno1.ponNombre("Pe" + finalNombre);
alumno1.ponNombre(alumno3.dameNombre());
```

2.3.1 Sobrecarga

El nombre del método y la lista de parámetros determinan unívocamente el método. Es decir, una misma clase (un mismo objeto) puede tener métodos tonel mismo nombre pero con distinto número de parámetros o con distinto tipo de los parámetros. Esto es lo que se llama **sobrecarga**. El sistema de ejecución determina cuál es el tipo de los parámetros que se pasan en una llamada concreta, y qué método hay que utilizar. Por ejemplo:

```
void ponAñoDeNacimiento (int año) {
  añoDeNacimiento = año;
}

void ponAñoDeNacimiento (String año) {
  añoDeNacimiento = Integer.parseInt(año);
}
```

2.3.2 Paso de parámetros

Cuando se llama a un método, se copian los valores de los argumentos en los parámetros. Es lo que se conoce en programación como paso por valor. Los cambios que el método haga sobre el parámetro no afectan al valor original del atributo. Por ejemplo:

```
void cambiaParametro (int i) {
  i++;
}
```

Si llamamos a este método:

```
int i = 1;
ejemplo.cambiaParametro(i);
System.out.println(i);
```

Imprimirá un 1.

Sin embargo, ¿qué pasa si el argumento es una referencia a un objeto? El valor que pasamos como parámetro es el valor de la referencia. Es decir, la referencia no se podrá cambiar. Sin embargo, sí que se puede cambiar el contenido del objeto mismo. Con este comportamiento hay que tener cuidado porque provoca muchos errores.
Por ejemplo:

```
class Alumno {

  String nombre;
  Alumno parejaPracticas;
```

```
    void ponPareja(Alumno pareja) {
      parejaPracticas = pareja;
      parejaPracticas.ponNombre(nombre); // Error - cuidado!!
    }
    void ponNombre(String nuevoNombre) {
      nombre = nuevoNombre;
    }
    String dameNombre() {
      return nombre;
    }

    public static void main (String[] args) {
      Alumno alumno1 = new Alumno();
      Alumno alumno2 = new Alumno();
      alumno1.ponNombre("Pedro");
      alumno2.ponNombre("Ana");
      alumno1.ponPareja(alumno2);

      System.out.println (alumno2.dameNombre());
    }
  }
```

¿Qué imprimirá este programa?

Lo que ha sucedido es lo siguiente. Tanto `alumno2` como `alumno1.parejaPracticas` apuntan al mismo objeto `Alumno`. Al pasar `alumno2` como argumento al método `ponPareja`, el valor de `alumno2` no se modifica (es decir, el valor de la dirección apuntada). Sin embargo, el contenido apuntado por `alumno2` está ahora también apuntado por el atributo `parejaPracticas` de `alumno1`. Si voluntariamente o por descuido modificamos cualquier campo de ese atributo, el resultado es que se modificará también el contenido de `alumno2`. Este comportamiento da lugar a veces a errores que son muy difíciles de localizar:

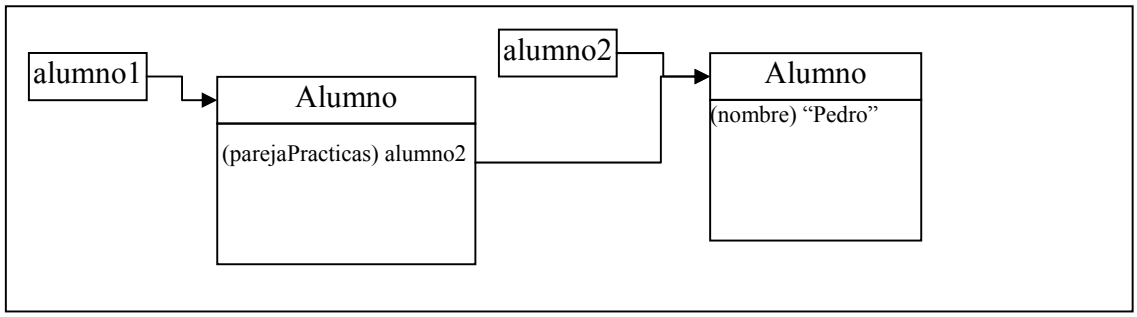

Figura 1: Ilustración de paso de parámetros

2.4 Cuerpo del método

El cuerpo del método es la parte rodeada de llaves que indica las acciones realizadas por el método. Contiene:

- Una serie de instrucciones, cada una de ellas separada por un punto y coma, que se ejecutan en secuencia empezando por la primera
- Una sentencia return en la que devolver el valor de salida (si el método no devuelve ningún valor, esta sentencia puede omitirse)
- Se pueden declarar variables locales dentro del cuerpo del método, que se pueden utilizar desde el momento de su declaración hasta el final del método
- Los parámetros que se declaran en la cabecera del método se pueden utilizar en el cuerpo del método como si fueran variables normales

Por ejemplo:

```
class Intercambio {

  void intercambiaValores(int x, int y) {
    int t;  // variable local para almacenamiento temporal
    t = x;
    x = y;
    y = x;
  }
}
```

3 Constructores

El constructor es un método especial que se llama cuando se crea un objeto. El objetivo del constructor suele ser inicializar el objeto.

Recordad cómo se crea un objeto:

```
alumno1 = new Alumno();
```

- con `new` se reserva la memoria necesaria en el heap
- con el constructor `Alumno()` se inicializa el objeto
- con el operador de asignación = se asigna la dirección del objeto a la referencia

El constructor es un método que tiene el mismo nombre que la clase, que puede tener cero, uno o varios parámetros, y que no devuelve ningún resultado.

Java llama al constructor de la clase cuando se crea el objeto. Todas las clases necesitan al menos un constructor.

Si el programador no define ningún constructor, el compilador se da cuenta y crea un Alumno() que hemos estado utilizando hasta ahora es el constructor por defecto. El constructor por defecto inicializa todos los atributos del objeto creado (a cero los numéricos, a null las referencias).

Se pueden definir varios constructores distintos para una misma clase, siempre que tengan distinta lista de parámetros. Es decir, se puede sobrecargar el constructor. En la llamada al constructor, según los argumentos concretos que se le pasen, el sistema de ejecución sabrá qué constructor usar.

Por ejemplo:

```
public class Coche {
  String matricula;
  String fabricante;
  String modelo;
  String color;

  public Coche (String matricula) {
    this.matricula = matricula;
  }

  public Coche (String matricula, String fabricante,
                String modelo,String Color) {
    this.matricula = matricula;
    this.fabricante = fabricante;
    this.modelo = modelo;
    this.color = color;
  }
}

public class PruebaCoche {
  public static void main (String[] args) {
    Coche c1 = new Coche ("3468 CXV");
    Coche c2 = new Coche ("4462 BIG", "BMW", "525i", "Negro");
    Coche c3 = new Coche ();   // ERROR!!
}
```

Hay que tener cuidado: hemos dicho que el compilador detecta si no hay ningún constructor definido, y en ese caso define el constructor por defecto, sin argumentos. Sin embargo, si hay algún constructor definido, no creará ninguno por defecto. En este caso, por ejemplo, no existe ningún constructor sin argumentos de la clase Coche. Si nos hace falta, tendremos que declararlo aparte.

Hay que saber, además, que no hace falta inicializar todos los atributos de la clase dentro del constructor. Podemos inicializar sólo algunos, como en el ejemplo. En ese caso, el resto de atributos se inicializarán a su valor por defecto (cero para números, null para referencias).

Tema 5: Estructuras de control

Contenidos

1. Introducción
2. Estructuras de selección
3. Bucles

1 Introducción

Dentro del cuerpo de los métodos se ejecutan sentencias (instrucciones) que determinan el comportamiento del programa. Hemos visto también que dichas sentencias se ejecutan en secuencia.
Las sentencias que hemos visto hasta ahora eran sencillas (asignación o return).

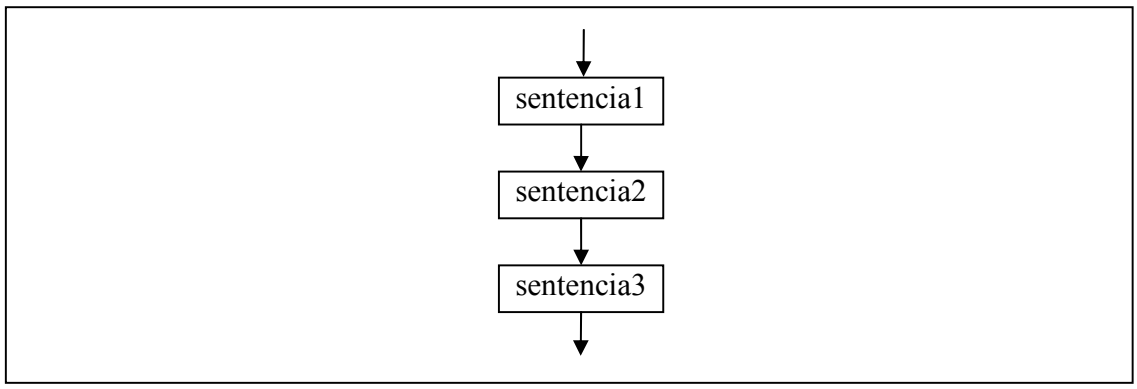

Figura 1: Ejecución en secuencia

Sin embargo, existen también sentencias complejas que permiten el control del flujo de ejecución.

2 Estructuras de selección

Las estructuras de selección permiten, en función del valor lógico de un selector, ejecutar un bloque de sentencias u otro.

2.1 Sentencias `if` e `if-else`

La sentencia `if` es la sentencia básica de selección. Tiene la forma:

```
if (condición) {
   sentencias
}
```

donde *`condición`* es una expresión booleana, y *`sentencias`* representa un bloque de sentencias o una sentencia única. Si es una sentencia única, se pueden quitar las llaves.

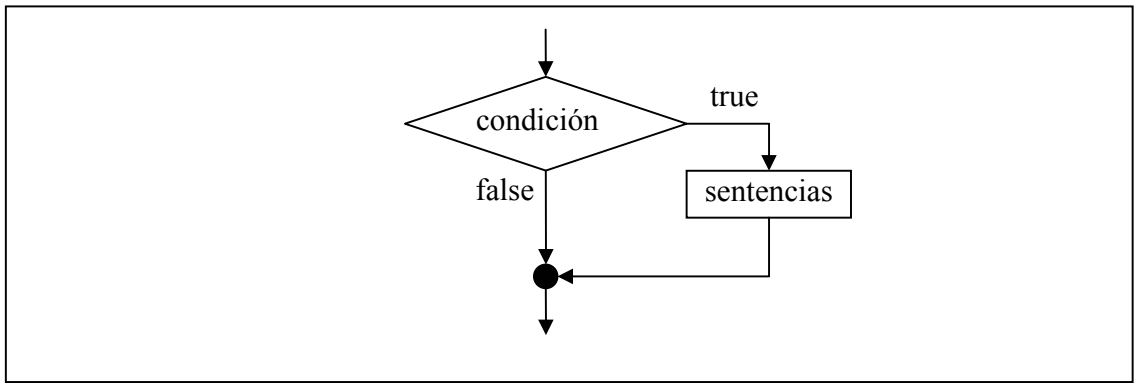

Figura 2: Sentencia if

El significado de la sentencia `if` es que, si la condición es cierta (si el valor de la expresión es `true`) se ejecutará el bloque de sentencias que contiene. Si la condición es falsa, dicho bloque de sentencias no se ejecutará. Gráficamente se puede ver en la Figura 2.

Opcionalmente, se puede añadir una parte `else` a la sentencia `if`, indicando otro bloque de sentencia a ejecutar si la condición es falsa.

```
if (condición) {
   sentencias1
} else {
   sentencias2
}
```

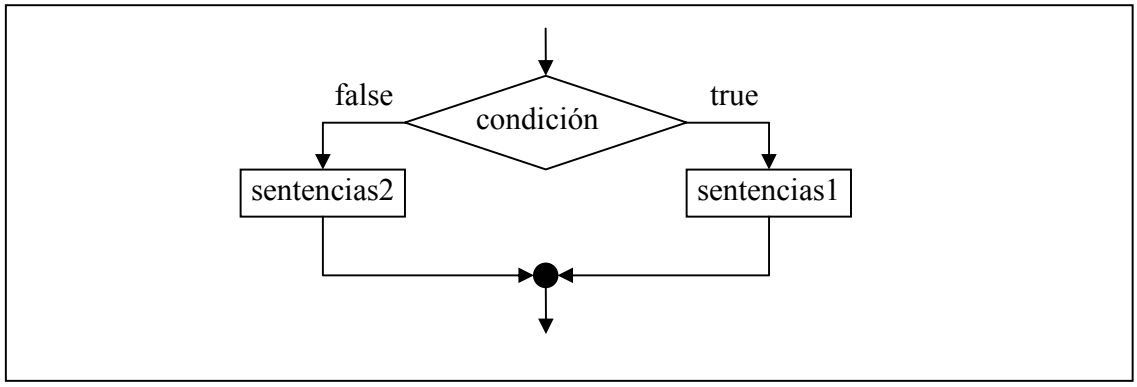

Figura 3: Sentencia if-else

A continuación se muestra un ejemplo de la estructura `if-else`:

```
public class IfElse {

  public void pruebaIfElse (int a, int b) {
    if (a>b) {
      System.out.println("El número mayor es " + a);
    } else {
      System.out.println("El número mayor es " + b);
    }
  }
```

```
  public static void main (String[] args) {
    IfElse ie = new IfElse();
    ie.pruebaIfElse(3,7);
    ie.pruebaIfElse(7,6);
  }
}
```

3 Bucles

Son también llamadas sentencias de iteración. La base de todas ellas es que un bloque de sentencias se repite hasta que una condición se evalúa a `false`.

3.1 Bucle `while`

La sintaxis de esta sentencia es:

```
while (condición) {
   sentencias
}
```

La condición es una expresión booleana que se evalúa al principio del bucle y antes de cada iteración de las sentencias. Si la condición es verdadera, se ejecuta el bloque de sentencias, y se vuelve al principio del bucle. Si la condición es falsa, no se ejecuta el bloque de sentencias, y se continúa con la siguiente sentencia del programa. Al igual que en caso de la sentencia `if`, si en el bloque de sentencias tenemos una sola sentencia, podemos quitar las llaves.

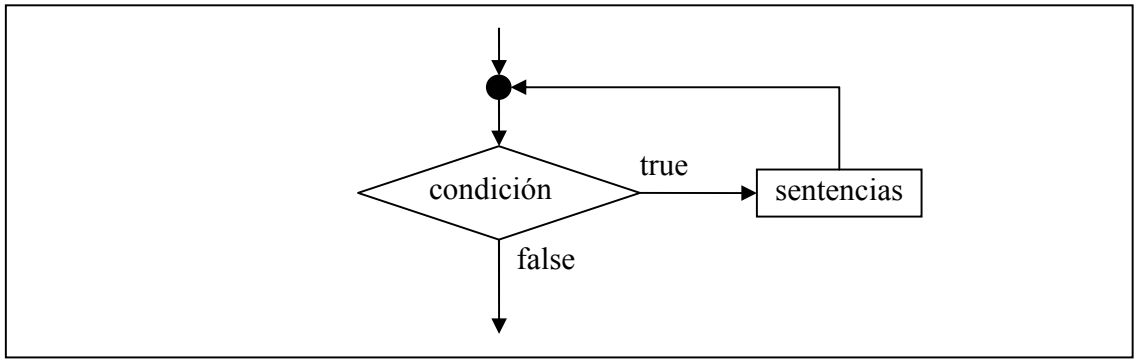

Figura 4: Sentencia while

Si la condición es falsa desde un principio, entonces el bucle nunca se ejecuta. Por eso se dice que el bucle `while` se ejecuta cero o más veces.

Si la condición nunca llega a ser falsa, tenemos un bucle infinito.

A continuación se muestra un ejemplo de un bucle `while`:

```
public class Primo {

  public static void main (String[] args) {
    int numero;
    int divisor;
```

```
    boolean esPrimo;

    numero = 101;
    divisor = 2;
    esPrimo = true;

    while ((divisor < numero /2) && esPrimo) {
      if (numero % divisor == 0)
        esPrimo = false;
      divisor++;
    }
    System.out.println("El numero " +numero);
    if (esPrimo)
      System.out.println(" es primo.");
    else
      System.out.println(" no es primo.");
  }
}
```

3.2 Bucle do-while

El bucle do-while es una sentencia muy parecida a while. El bloque de sentencias asociado se repite mientras se cumpla una condición. La diferencia con el bucle anterior es que la condición se comprueba después de ejecutar el bloque de sentencias. Es decir, el bloque se ejecuta siempre al menos una vez.

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

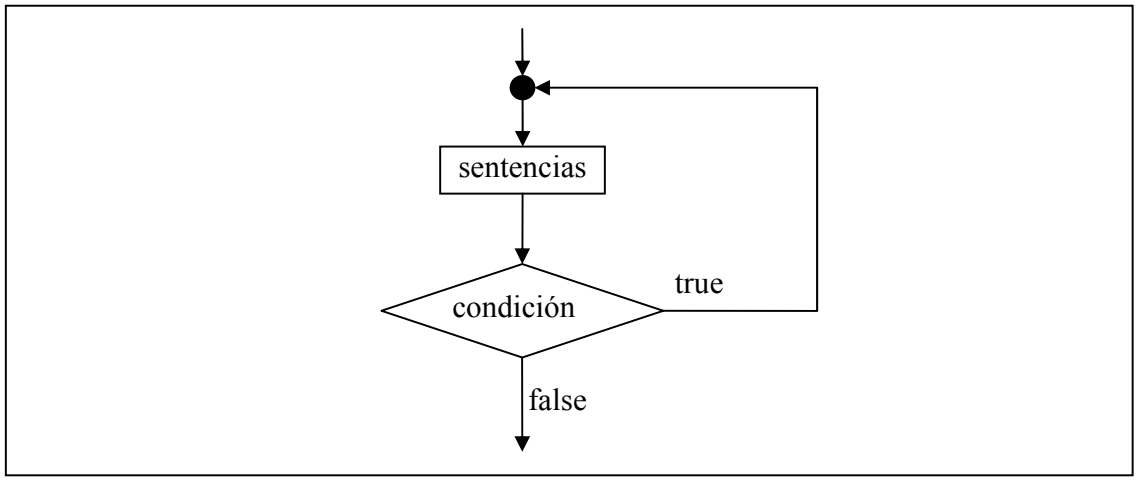

Figura 5: Sentencia do-while

Por ejemplo:

```
import java.io.*;

public class PedirNumero {
```

```
  public static void main (String[] args){
    int numero;
    String linea;
    BufferedReader teclado = new BufferedReader(
                 new InputStreamReader(System.ini));
    do {
      System.out.println("Introduce un número entre 1 y 100:");
      linea = teclado.readLine();
      numero = Integer.parseInt(linea);
    } while (numero < 0 || numero > 100);
    System.out.println("El numero introducido es: " + numero);
  }
}
```

3.3 Bucle `for`

La sintaxis del bucle `for` es la siguiente:

```
for (inicialización ; condición ; actualización) {
   sentencias
}
```

- La inicialización se realiza sólo una vez, antes de la primera iteración
- La condición se comprueba cada vez antes de entrar al bucle. Si es cierta, se entra. Si no, se termina.
- La actualización se realiza siempre al terminar de ejecutar la iteración, antes de volver a comprobar la condición

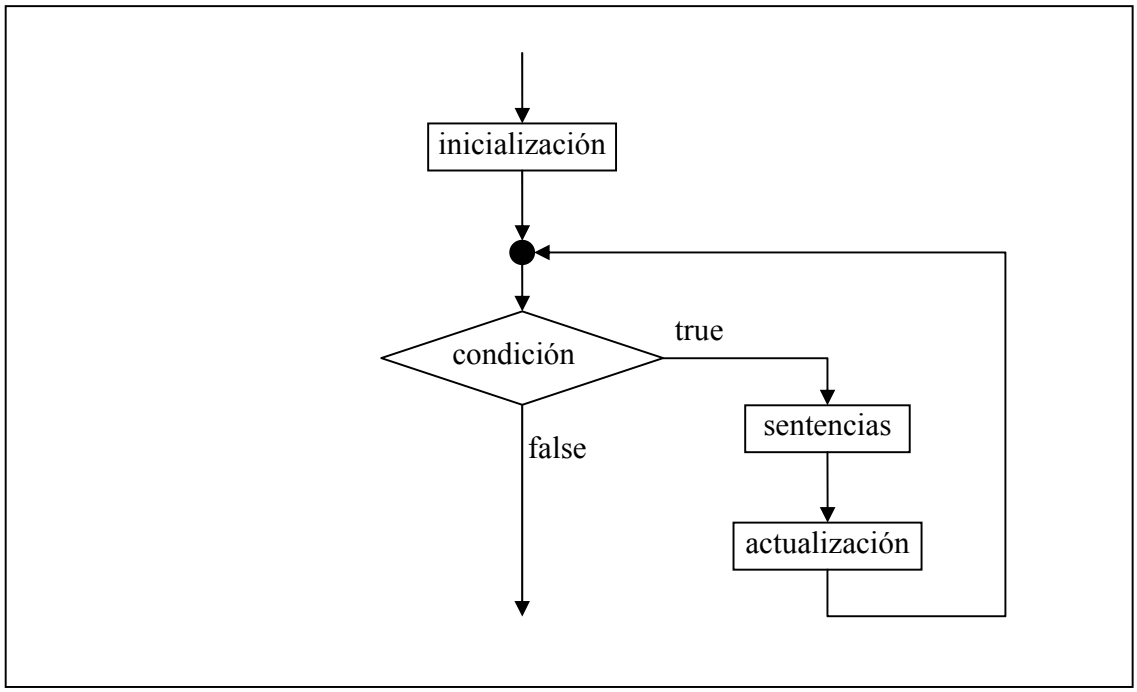

Figura 6: Sentencia for

Por ejemplo:

```
public class ListaCaracteres {

  public static void main (String[] args) {
    for (char c = 0; c < 128; c++) {
      if (c != 26)    // caracter para borrar pantalla
        System.out.println("Valor: " + (int)c
                                    + "\t Carácter: " + c);
  }
}
```

Tema 6: Ampliación de clases

Contenidos

1. Elementos de clase: static
2. Derechos de acceso
3. Paquetes
4. Clases envoltorio
5. La clase String

1 Elementos de clase: static

Hasta ahora, los atributos y métodos que hemos visto se definen en la clase pero se utilizan siempre sobre un objeto concreto. Por ejemplo, si queremos utilizar el método `ponNombre` definido en `Alumno` tendremos que llamarlo sobre un objeto de la clase `Alumno`: `alumno1.ponNombre("Pepe").`

Sin embargo, hay métodos y atributos que pertenecen a la clase, es decir, no se los invoca sobre un objeto concreto, sino que están disponibles siempre, sin necesidad de crear previamente un objeto de la clase. Estos atributos y métodos se llaman *elementos de clase* y están precedidos por la palabra clave reservada `static`.

Ya hemos visto un par de ejemplos:

- `public static void main (String[] args)`
  El método main de una clase es estático porque el intérprete (la máquina virtual java - JVM) lo invoca al principio de todo, antes de haber podido crear ningún objeto de la clase. Es un método estático "especial" porque lo invoca la JVM (nosotros dentro de nuestro código nunca llamamos a main).
- `Integer.parseInt(String entero)`
  Este es un método estático de la clase `Integer` (una clase envoltorio que sirve para representar a los objetos como enteros – se verá en secciones posteriores de este tema). Los métodos estáticos se invocan poniendo por delante el nombre de la clase y no el nombre de un objeto (ya que no se llaman sobre ningún objeto concreto).
- `System.out`
  Es un atributo estático de la clase `System` que representa la salida estándar de la máquina (que es la pantalla, si no se redirige).
- `System.in`
  Es un atributo estático de la clase `System` que representa la entrada estándar de la máquina (que es el teclado, si no se redirige).

- Los atributos y métodos de clase estáticos son compartidos por todas las sentencias de la clase. Si se modifican dentro de un método, el cambio es visible en todas partes. Es decir, son variables "globales" a diferencia de las variables que habíamos visto hasta

ahora, que eran locales. Esto es muy peligroso y `en` general desaconsejado, salvo para representar constantes (que llevarán además la palabra reservada `final`).

Si un método estático (por ejemplo `main`) intenta utilizar directamente atributos y métodos de la clase (es decir, sin crear previamente un objeto y llamarlos sobre el objeto) dará un error, ya que el método estático se llama sin crear un objeto, y los no-estáticos necesitan un objeto. Por lo tanto, el `main` debe crear primero el objeto y luego hacer las llamadas. Ya veréis que este error se comete mucho al principio, y el compilador os dará el mensaje correspondiente.

2 Derechos de acceso

Java proporciona un mecanismo para definir distintos derechos de acceso tanto a la clase como a sus elementos (atributos y métodos).

- Acceso privado. Se representa con la palabra reservada `private`. Los elementos privados sólo pueden ser usados dentro de la clase que los define. No son accesibles desde el exterior.
- Acceso de paquete. Éste es el acceso por defecto, es decir, si no se utiliza ninguna palabra reservada. Los elementos son accesibles desde dentro del paquete que los define. Veremos en la próxima sección qué es un paquete.
- Acceso protegido. Se representa con la palabra reservada `protected`. Estos elementos pueden ser usados en la clase que los define, en las de su paquete y en las que heredan de ella. La herencia en Java se tratará en detalle en próximos temas.
- Acceso público. Se representa con la palabra reservada `public`. Se puede acceder a este elemento desde cualquier clase.

Normalmente, los atributos de una clase serán privados, para que no se pueda modificar descontroladamente el estado de un objeto desde fuera. Para modificar los atributos se utilizarán los métodos públicos definidos por el programador, como `ponNombre`.

Los constructores y los métodos principales serán normalmente públicos. Puede ser que el programador defina algún método auxiliar para hacer un cálculo intermedio utilizado por otro método de la clase. En ese caso, el método auxiliar puede ser privado, porque sólo será accedido desde dentro de la clase.

Ejemplo:

```
public class Alumno {

  private String nombre;              //atributos privados
  private String apellidos;
  private int añoDeNacimiento;
  ...
  public Alumno(int nia) {             //constructor público
    ...
  }
```

```
    public void ponNombre (String nombre) { //métodos públicos
    ...
    }
}
```

3 Paquetes

Un paquete es una forma de agrupar clases relacionadas entre sí. Los paquetes representan bibliotecas de clases.

3.1 Importación de paquetes

Cuando nuestro programa necesita usar alguna clase de la biblioteca (del paquete) necesita importarla. Esto se hace con la sentencia import que se coloca al principio del programa, antes de comenzar con la definición de la clase:

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

Por ejemplo, con esta sentencia estamos importando el paquete de entrada/salida (para leer del teclado, de fichero, etc.) que forma parte de la distribución estándar de Java, y que utilizamos, por ejemplo, en el programa que leía del teclado. Si importamos un paquete, podemos usar directamente las clases que están dentro de ese paquete. Por ejemplo, la clase `BufferedReader` está dentro del paquete `java.io`:

```
BufferedReader teclado = new BufferedReader (...);
```

Si no importamos el paquete y queremos utilizar la clase, tenemos que usar el nombre cualificado completo:

```
java.io.BuffereredReader teclado ...
```

3.2 Creación de paquetes

Un programa en la vida real puede tener cientos o miles de clases. Hasta ahora, se han colocado todas las clases en el mismo directorio, y desde ese directorio habéis llamado al intérprete de java (comando `java`), y el intérprete ha encontrado la clase. Pero si tenéis muchas clases, querréis organizarlas de alguna forma.

Imaginaos que tenéis que hacer un programa para la gestión de matrículas de los alumnos, que consta de muchas clases java (`Alumno.class`, `Asignatura.class`, `Administración.class`, `Curso. class`, etc etc). Además, queremos pasarle al programa muchas pruebas, y las escribimos en distintos ficheros java (`Prueba1.class`, `Prueba2.class`, `Prueba3.class`, `Prueba4.class`, `Prueba5.class`, etc, etc).

En vez de tenerlo todo mezclado, lo agruparemos en dos paquetes: el paquete matricula y el paquete pruebas.

Para las clases que vayan a ir en el paquete matricula debo incluir la siguiente línea al principio del fichero. Por ejemplo, si queremos que la clase Alumno pertenezca al paquete matricula, escribiríamos al principio del fichero `Alumno.java`:

```
package matricula;
...
public class Alumno {
  ...
}
```

Lo mismo se haría si, por ejemplo, programamos la clase Prueba1 dentro del paquete pruebas:

```
package pruebas;
...
public class Prueba1 {
  ...
}
```

De esta forma declaramos que estas clases van a pertenecer al paquete. Para usar las clases de estos paquetes dentro de otras clases, se usa la sentencia `import`, de la misma forma que se usa para paquetes definidos directamente en el lenguaje java como `java.io`.

Por ejemplo, la clase `Prueba1` va a necesitar las clases del paquete `matricula`. Entonces, las primeras líneas del fichero `Prueba1.java` serían algo así como:

```
package pruebas;
import matricula.*;

public class Prueba1 {
  ...
}
```

Además de darles un nombre a los paquetes, conviene darles una estructura física a los ficheros de las clases de un paquete. La forma más habitual es colocar todos los ficheros `.class` de un paquete en un subdirectorio con el mismo nombre que el paquete. Así, en nuestro ejemplo se pondría:

```
miDirectorio --- matricula --- Alumno.java
                |             |
                |             -- Alumno.class
                |             |
                |             -- Asignatura.java
                |             |
                |             -- Asignatura.class
                |             |
                |             -- Curso.java
                |             |
                |             -- Curso.class
                |             |
                |             -- ...
                |
                |-- pruebas  --- Prueba1.java
                               |
                               -- Prueba1.class
                               |
                               -- Prueba2.java
                               |
                               -- Prueba2.class
                               |
                               -- ...
```

Para compilar, desde `miDirectorio`, haréis, por ejemplo:

```
> javac matricula/*.java
> javac pruebas/*.java
> java prueba.Prueba1
```

4 Clases envoltorio

Son unas clases que permiten representan a los tipos primitivos como si fueran clases. Las clases envoltorio tienen dos utilidades principales:

- Definen métodos estáticos útiles, por ejemplo, para transformación de formatos, etc.
  ```
  String linea = teclado.readLine();
  int numero = Integer.parseInt(linea);
  ```
- Permiten representar valores de tipos primitivos como si fueran objetos. Por ejemplo:
  ```
  Integer miEntero = new Integer(1024);
  Integer otroEntero = new Integer(2048);
  boolean res = miEntero.equals(otroEntero);
  ```

Otro ejemplo:

```
int numero = 37;
System.out.println("Valor entero: " + numero);
```

El operador + está definido para concatenar 2 Strings, y el segundo operando que le estamos pasando es un entero. ¿Cómo puede funcionar esto? Lo que sucede aquí es que el sistema java aplica automáticamente el envoltorio a la variable de tipo `int` (está haciendo algo así como `Integer envoltorioEntero = new Integer (numero);`) y después llama al método `toString` (definido en la clase `Integer`), del objeto envoltorio generado

(equivalente a hacer `String stringEntero = envoltorioEntero.toString();`). El método `toString` está definido para todas las clases envoltorio y genera una representación en forma de `String` del valor de tipo primitivo contenido dentro del envoltorio sobre el que se llama a la función.

Las clases envoltorios definidas son:

Envoltorio	Tipo primitivo
`Boolean`	`boolean`
`Character`	`char`
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`

5 La clase `String`

La clase `String` es una de las clases predefinidas de java más útiles. Un objeto de la clase String representa una cadena de caracteres.

Ya se ha visto en temas previos aspectos básicos de los `String`, como que sus literales se escriben entre comillas, y que el operador + sirve para concatenar Strings.

Otros métodos interesantes de la clase String son:

- `equals`: compara los contenidos de los Strings, devuelve `true` si los contenidos son iguales, y `false` en caso contrario

```
String a = "Pedro";
String b = "Pedro";
System.out.println(a==b);        // false: compara referencias
System.out.println(a.equals(b)); // true: compara contenido
```

- `compareTo`: compara los contenidos de los Strings, devuelve `0` si los contenidos son iguales, un valor positivo si el objeto sobre el que se llama al método (referenciado por this) es lexicográficamente mayor que el que se le pasa como parámetro, y un valor negativo si el objeto sobre el que se llama al método es lexicográficamente menor que el que se le pasa como parámetro

```
String a = "Pedro";
String b1 = "Carlos";
String b2 = "Pedro";
String b3 = "Pepe";
int comp1 = a.compareTo(b1); // comp1 == 0
int comp2 = a.compareTo(b2); // comp2 > 0
int comp3 = a.compareTo(b3); // comp3 < 0
```

- `length`: devuelve la longitud del `String` (el número de caracteres)

- `charAt`(índice): devuelve el carácter localizado en la posición especificada. Los caracteres dentro de un `String` se numeran del 0 a N-1, siendo N la longitud `del String`

Hay que tener siempre cuidado de llamar a estos métodos sólo sobre referencias a String distintas de `null`. Si hago algo así:

```
String a;
int i = a.length();
```

me va a dar error `NullPointerException`. ¿Por qué?

6 El API de Java

La descripción de todas las clases predefinidas en java y de sus métodos se encuentra en la documentación del API (Application Programming Interface).

Esta documentación la podéis encontrar en `http://java.sun.com`, o la tendréis en vuestra máquina si os habéis instalado el JDK. Además, está de forma local en el departamento en:

`http://www.it.uc3m.es/java/InfoAdicional/docs/api/`

Siempre que programéis, es importante tener abierta la página de la documentación.

Tema 7: Extensión de clases

Contenidos

1. Excepciones
2. Conceptos básicos de herencia

1 Excepciones

En todo programa se pueden producir errores o situaciones excepcionales. Cuando se produce una de estas situaciones, el programa puede terminar o puede tratar la excepción, ya que algunas de ellas son previsibles.

En Java, las excepciones se representan como objetos de la clase Exception. Más concretamente, las excepciones se organizan según una estructura jerárquica definida en Java, donde Exception es la clase más genérica, y el resto de tipos de excepciones heredan de ella:

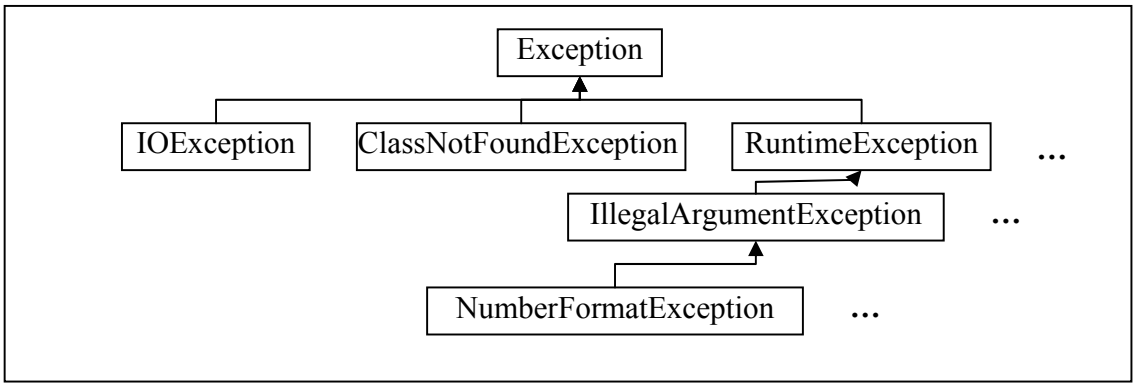

Figura 1: Jerarquía de excepciones

Incluso podríais definiros vosotros vuestra propia clase de excepciones.

Cuando un método lanza una excepción, quiere decir que detecta que se ha producido una situación anómala, pero no sabe que hacer con ella. En ese caso, el método lanza la excepción y termina, propagando la excepción al método que le llamó, que quizá sepa qué hacer en ese caso.

Veamos de nuevo un programa que apareció en el tema 5:

```
import java.io.*;

public class PedirNumero {

  public static void main (String[] args)
          throws IOException{
    int numero;
    String linea;
```

```
        BufferedReader teclado = new BufferedReader(
                    new InputStreamReader(System.in));
        do {
          System.out.println("Introduce un número entre 1 y 100:");
          linea = teclado.readLine();
          numero = Integer.parseInt(linea);
        } while (numero < 0 || numero > 100);
        System.out.println("El numero introducido es: " + numero);
      }
    }
```

El método `readLine()` de la clase `BufferedReader` puede lanzar la excepción `IOException` cuando hay un error de entrada por el teclado (por ejemplo, el teclado está desconectado). Si veis en la documentación del API (`http://www.it.uc3m.es/java/InfoAdicional/docs/api/`) la declaración de ese método, es la siguiente:

```
public String readLine() throws IOException
```

La excepción que un método lanza se propaga al método llamante, y así sucesivamente. La cadena de llamadas en este caso es:

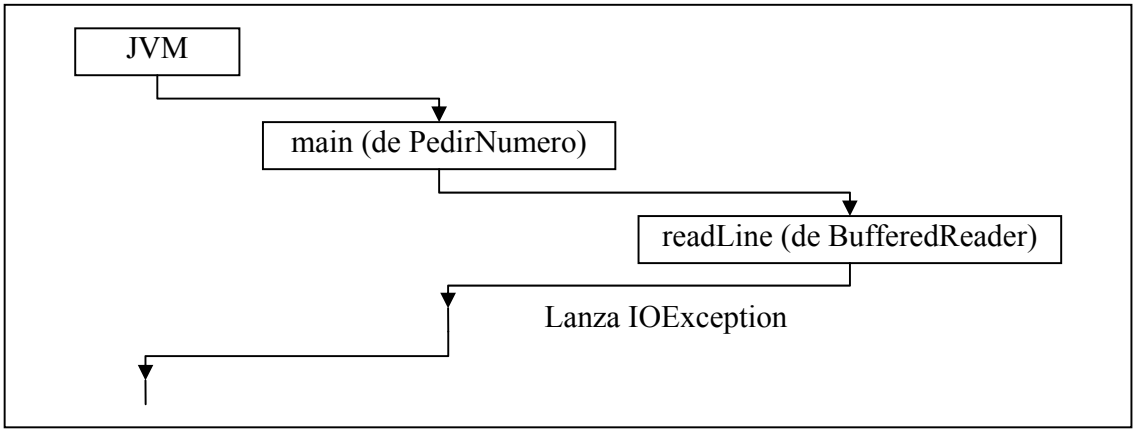

Figura 2: Cadena de llamadas

Es decir, la excepción lanzada por readLine() será procesada por el main, o bien será relanzada del main a la Java Virtual Machine, que terminará la ejecución del programa e indicará qué excepción se produjo.

También se podría haber llamado a readLine() desde un método normal de una clase, distinto del main. Por ejemplo, imaginad que añadimos a la clase Alumno un método

```
public void pideNombre() throws IOException {…}
```

que pide por teclado el nombre del alumno, y que en una clase Prueba tenemos un método main en el que creamos un objeto de la clase Alumno y llamamos a su método pideNombre. En este caso, la cadena de llamadas sería:

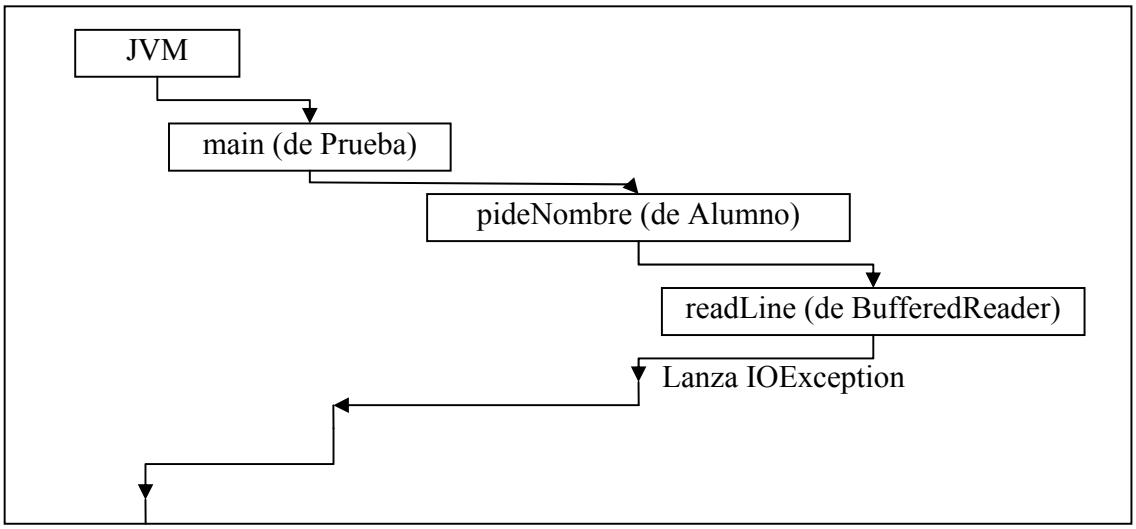

Figura 3: Otra cadena de llamadas

Cuando un método llama a otro método que puede lanzar una excepción, el método llamante puede:

- Capturar y manejar la excepción, o bien
- Relanzarla

Además, un método de usuario puede lanzar una excepción cuando él mismo detecte una situación anómala.

Un caso especial son las excepciones del subrupo RuntimeException. Representan errores en tiempo de ejecución que se pueden producir en cualquier momento. Por ejemplo, NullPointerException (cuando se intenta acceder a los campos de una referencia que está a null). Este tipo de excepciones son lanzadas por Java en tiempo de ejecución. Los programas de usuario no necesitan capturarlas ni relanzarlas, se propagarán automáticamente hasta la JVM y provocarán la terminación del programa.

1.1 Lanzamiento de excepciones

Vamos a redefinir el método ponGrupo de la clase Alumno para que compruebe si los valores que se le pasan como argumento están en el rango permitido:

```
public void ponGrupo(String grupo, char horario)
    throws Exception {
  if (grupo == null || grupo.length() == 0)
     throw new Exception ("Grupo no válido");
  if (horario != 'M' && horario != 'T')
     throw new Exception ("Horario no válido");

  this.grupo = grupo;
  this.horario = horario;
}
```

El lanzamiento de una excepción se realiza con la sentencia throw, que va seguida de un objeto de la clase Exception (o de una de sus clases derivadas). La clase concreta de la excepción debe coincidir con la clase especificada en la cabecera del método.

El objeto de la clase Exception se crea como siempre son una llamada a new, seguida del nombre del constructor. Existe un constructor sin argumentos, y otro que recibe como argumento un mensaje explicativo del error ocurrido. Se recomienda usar ese constructor, para tener más información sobre el error.

1.2 Captura de excepciones

Si el método llamante decide capturar y manejar una excepción en vez de relanzarla, lo hace con la sentencia try – catch – finally.

```
try {
      sentencias que pueden producer la excepción
} catch (ClaseException1 e1) {
      sentencias de manejo de excepciones tipo ClaseException1
} catch (ClaseException2 e2) {
      sentencias de manejo de excepciones tipo ClaseException2
} ... {
} finally {
      sentencias de finalización
}
```

Las sentencias que son susceptibles de producir una excepción se engloban dentro del try. En la parte del catch se comprueba la clase de la excepción. Si coincide con `ClaseException1` se ejecutan lasa sentencias asociadas, si no, se sigue probando con las otras clases de excepciones (en el caso de que dentro del código se puedan producir distintos tipos de excepciones). La parte finally es opcional y se ejecutará en cualquier caso.

Como ejemplo, vamos a redefinir la clase PedirNumero para que maneje la excepción en lugar de relanzarla:

```
import java.io.*;

public class PedirNumero2 {

  public static void main (String[] args) {
    int numero = -1;
    String linea;
    BufferedReader teclado = new BufferedReader(
                new InputStreamReader(System.in));
    do {
      try {
        System.out.println("Introduce un número entre 1 y 100:");
        linea = teclado.readLine();
        numero = Integer.parseInt(linea);
      } catch (IOException e) {
        System.out.println("Error al leer del teclado");
      }
    } while (numero < 0 || numero > 100);
```

```
    System.out.println("El numero introducido es: " + numero);
  }
}
```

2 Conceptos básicos de herencia

La herencia es la capacidad de extender clases, de forma que la nueva clase hereda todo el comportamiento y código de la clase extendida. En este tema se pueden usar las siguientes denominaciones:

- Clase original, clase base, clase padre, superclase
- Clase derivada, clase hija, subclase

Ejemplos de herencia (clase base → clase derivada):

- clase Conjunto → clase ConjuntoOrdenado
- clase Grupo → clase GrupoAbeliano
- clase Mamífero → clase Perro, clase Gato
- clase Persona → clase Alumno

La herencia realiza la rlación "es-un": un perro es un mamífero, un gato es un mamífero, etc.

La sintaxis para declarar clases derivadas es:

```
class ClaseDerivada extends ClaseBase { ... }
```

Veamos el siguiente ejemplo:

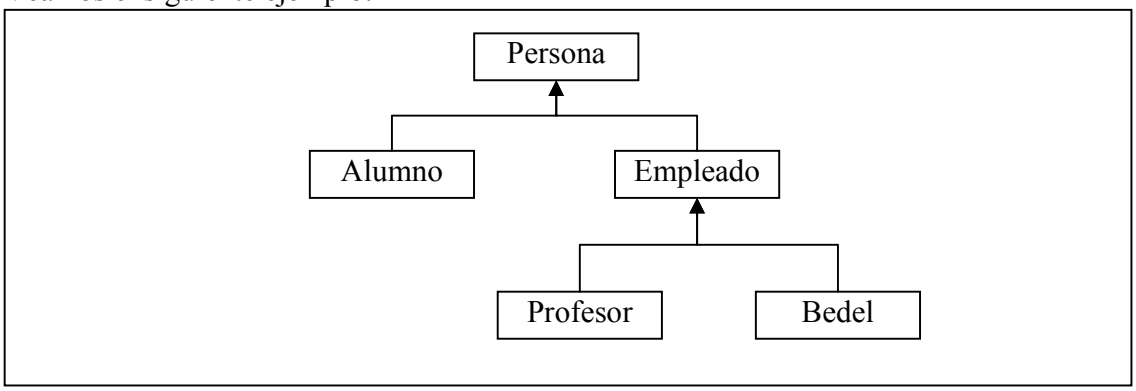

Figura 4: Jerarquía de herencia

La declaración de estas clases seguiría la forma:

```
class Persona { ... }
class Alumno extends Persona { ... }
class Empleado extends Persona { ... }
class Profesor extends Empleado { ... }
class Bedel extends Empleado { ... }
```

Por ejemplo, en el fichero Persona.java tendríamos:

```
public class Persona {
  protected String nombre;
  protected String apellidos;
  protected int anyoNacimiento;
```

```
    public Persona () {
    }

    public Persona (String nombre, String apellidos,
                    int anyoNacimiento){
      this.nombre = nombre;
      this.apellidos = apellidos;
      this.anyoNacimiento = anyoNacimiento;
    }

    public void imprime(){
      System.out.print("Datos Personales: " + nombre
                                    + " " + apellidos + " ("
                                    + anyoNacimiento + ")");
    }
}
```

En el fichero Alumno.java tendríamos:

```
public class Alumno extends Persona {
  protected String grupo;
  protected char horario;

    public Alumno() {
    }

    public Alumno (String nombre, String apellidos,
                   int anyoNacimiento) {
      super(nombre, apellidos, anyoNacimiento);
    }

    public void ponGrupo(String grupo, char horario)
        throws Exception {
      if (grupo == null || grupo.length() == 0)
         throw new Exception ("Grupo no válido");
      if (horario != 'M' && horario != 'T')
         throw new Exception ("Horario no válido");

      this.grupo = grupo;
      this.horario = horario;
    }

    public void imprimeGrupo(){
      System.out.print(" Grupo " + grupo + horario);
    }
}
```

Finalmente, supongamos que en el fichero Prueba.java tenemos:

```
public class Prueba {
   public static void main (String[] args) throws Exception{

     Persona vecina =
```

```
                    new Persona ("Luisa", "Asenjo Martínez", 1978);
        Alumno unAlumno = new Alumno ("Juan", "Ugarte López", 1985);
        unAlumno.ponGrupo("66", 'M');

        vecina.imprime();

        System.out.println();

        unAlumno.imprime();
        unAlumno.imprimeGrupo();
    }
}
```

La extensión de la clase provoca:

- la herencia del interfaz: la parte pública de la clase derivada contiene la parte pública de la clase base: la clase Alumno contiene el método imprime() (al revés no es cierto: la clase Pe3rsona no contiene el método ponGrupo ni el método imprimeGrupo)
- la herencia de la implementación: la implementación de la clase derivada contiene la de la clase base: al invocar los métodos de la clase base sobre el objeto de la clase derivada (unAlumno.imprime()) se produce el comportamiento esperado.

Tema 8: Herencia avanzada

Contenidos

1. Jerarquía de herencia
2. Compatibilidad de tipos
3. Reescritura
4. Constructores
5. Clases abstractas
6. Interfaces

1 Jerarquía de herencia

En Java, todas las clases están relacionadas en una única jerarquía de herencia. Una clase puede:

- heredar explícitamente de otra clase,
- o bien hereda implícitamente de la clase `Object` (definida en el núcleo de Java), si no hereda de otra.

Esto se cumple tanto para las clases definidas en Java como para las clases definidas por el usuario.

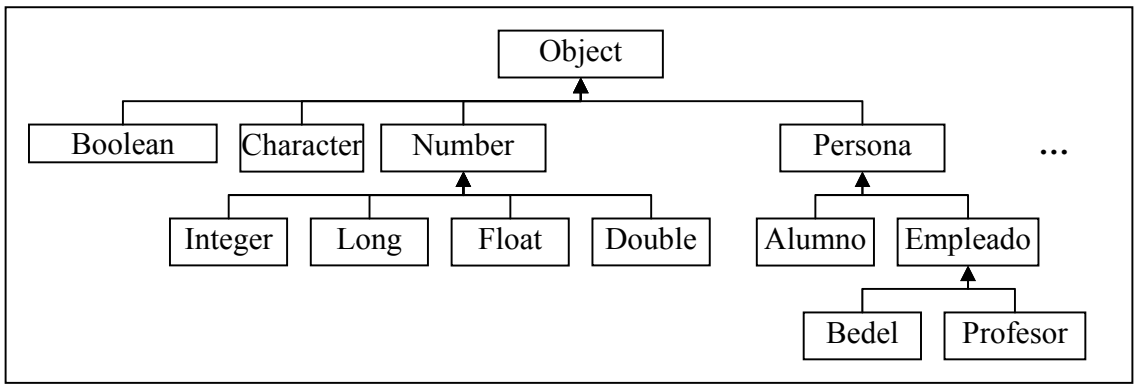

Figura 1: Jerarquía de herencia

2 Compatibilidad de tipos

Los objetos de las clases bases y derivadas tienen un cierto nivel de compatibilidad:

1. Compatibilidad hacia arriba (*upcasting*):
   Un objeto de la clase derivada siempre se podrá usar en el lugar de un objeto de la clase base (ya que se cumple la relación "es-un"):
   ```
   Persona p = new Alumno();
   ```
2. Compatibilidad hacia abajo (*downcasting*)
   Normalmente no es posible, ya que un objeto de la clase base no siempre es un objeto de la clase derivada (una Persona no siempre es un Alumno):
   ```
   Alumno a = new Persona(); // error
   ```

La compatibilidad hacia abajo sólo es posible en los casos en los que el objeto de la clase base realmente sea un objeto de la clase derivada. Estos casos se tendrán que indicar explícitamente con un casting (con una asignación explícita de la clase).

Veamos el ejemplo siguiente (basado en las clases `Persona` y `Alumno` presentadas en el tema anterior):

```
public class Prueba2 {
   public static void main (String[] args) throws Exception{
     Persona p1;
     Alumno a1 = new Alumno();
     p1 = a1;                    //conversión ascendente

     Alumno a2;

     a2 = p1; //error porque no hago conversión explícita

     //conversión descendente explícita - funciona
     a2 = (Alumno) p1;   //p1 referencia una instancia de Alumno

     Persona p2 = new Persona();
     Alumno a3;

     a3 = p2;   //da error de compilación

     //aunque hagamos conversión descendente explícita,
     //lanzará la excepción ClassCastException
     //porque p2 no es de la clase Alumno
     a3 = (Alumno) p2; //error

     Alumno a4 = new Persona(); //error
   }
}
```

Si no se sabe seguro si el objeto sobre el que se hace la conversión descendente explícita es realmente de la clase derivada, se puede comprobar con el predicado:
`objeto` **`instanceOf`** `clase`

Por ejemplo:

```
public Alumno comprueba (Persona p) {
  Alumno a;
  if (p instanceOf Alumno)
    a = (Alumno) p;
  return a;
}
```

3 Reescritura

Es la modificación de los elementos de la clase base dentro de la clase derivada. La clase derivada puede definir:

- un atributo con el mismo nombre que uno de la clase base
- un método con la misma signatura que uno de la clase base

(Se recuerda que la signatura de un método está formada por su nombre, el la clase del resultado devuelto, y el número y clase de sus parámetros).
Lo más usual cuando se produce reescritura es que se reescriba un método.

Tomemos como base el ejemplo mostrado en el Tema 7 con las clases `Persona` y `Alumno`. Supongamos que la clase `Alumno` reescribe el método imprime de la clase `Persona` (de la que hereda):

```
public class Alumno extends Persona {
  // el resto permanence igual
   public void imprime(){
     System.out.print("Datos Personales: " + nombre
                      + " " + apellidos + " ("
                      + anyoNacimiento + ")"
                      + " Grupo " + grupo + horario);
  }
}
```

Si adaptamos la clase de `Prueba` de la siguiente forma:

```
public class Prueba3 {
   public static void main (String[] args) throws Exception{

     Persona vecina =
                 new Persona ("Luisa", "Asenjo Martínez", 1978);
     Alumno unAlumno = new Alumno ("Juan", "Ugarte López", 1985);
     unAlumno.ponGrupo("66", 'M');

     vecina.imprime();

     System.out.println();

     unAlumno.imprime();
  }
}
```

Veremos que se llama al nuevo método reescrito. Pero la potencia del la reescritura es que se llama al método correcto, aunque nos estemos refiriendo al objeto `unAlumno` a través una referencia de tipo `Persona`. Esto sucede por un mecanismo llamado “ligadura dinámica”, que permite a Java detectar en tiempo de ejecución cuál es el método adecuado para llamar. Esto se refleja en el siguiente ejemplo:

```
public class Prueba4 {
   public static void pruebaImpresion(Persona p) {
     p.imprime();
   }
```

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

      Persona vecina =
                  new Persona ("Luisa", "Asenjo Martínez", 1978);
      Alumno unAlumno = new Alumno ("Juan", "Ugarte López", 1985);
      unAlumno.ponGrupo("66", 'M');

      pruebaImpresion(vecina);

      System.out.println();

      pruebaImpresion(unAlumno);
    }
}
```

Para reescribir métodos, puede ser útil usar la referencia a `super`.
Un objeto de la clase derivada contiene internamente un subobjeto de la clase base. A este subobjeto se le referencia mediante la palabra `super`. A través de la referencia a `super` se puede acceder explícitamente a métodos de la clase base.

Por ejemplo, podríamos haber escrito:

```
public class Alumno extends Persona {
  // el resto permanence igual
   public void imprime(){
     super.imprime();
     System.out.print(" Grupo " + grupo + horario);
  }
}
```

La reescritura de métodos es útil para:

- ampliar la funcionalidad de un método
- particularizar la funcionalidad de un método a la clase derivada

Si no se quiere que las clases derivadas sean capaces de modificar un método o un atributo de la clase base, se añade a ese método o atributo la palabra reservada `final`.

4 Constructores

Para la creación de un objeto se siguen los siguientes pasos:

1. Se crea su parte base
2. Se añade su parte derivada

Si la clase base del objeto hereda a su vez de otra, en el paso 1 se aplica el mismo orden de creación, hasta llegar a Object. Por ejemplo, en la creación de un objeto Alumno que hereda de Persona, los pasos son:

1. Se crea la parte correspondiente a Persona. Para ello:
   1.1. Se crea la parte correspondiente a Object
   1.2. Se añaden los elementos de Persona
2. Se añaden los elementos de Alumno

En el constructor de la clase derivada se realiza siempre una llamada al constructor de la clase base, y ésta es la primera acción del constr4uctor (aparece en la primera línea). Hay dos posibilidades:

1. Si no se indica explícitamente, Java inserta automáticamente una llamada a super() (el constructor de la clase base sin parámetros) en la primera línea del constructor de la clase derivada.

```
public Alumno (String nombre, String apellidos,
               int anyoNacimiento, String grupo, char horario) {
  // aquí inserta Java una llamada a super()
  this.nombre = nombre;
  this.apellidos = apellidos;
  this.anyoNacimiento = anyoNacimiento;
  this.grupo = grupo;
  this.horario = horario;
}
```

2. Indicándolo explícitamente:

```
public Alumno (String nombre, String apellidos,
               int anyoNacimiento, String grupo, char horario) {
  super(nombre, apellidos, anyNacimiento);
  this.grupo = grupo;
  this.horario = horario;
}
```

5 Clases abstractas

En ocasiones las clases bases se introducen sólo para ser heredadas, pero no se quiere instanciar directamente a objetos de esa clase.

Por ejemplo, en el programa de gestión de matrículas que estamos manejando, puede decidirse definir la clase Persona como factor común para otras clases, pero puede no quererse crear ningún objeto de la clase Persona, sino únicamente de Alumno, Profesor y Bedel. Para esto se usan las clases abstractas.

Una clase abstracta es aquélla que no se puede instanciar, pero sí heredar. En los programas, pueden existir referencias a clases abstractas, que en realidad apuntarán a objetos de clases derivadas de la clase absracta. Una clase abstracta se define con la palabra reservada abstract:

```
public abstract class Persona { // resto de la clase igual}
```

5.1 Métodos abstractos

Las clases abstractas suelen usarse casi siempre para representar clases con implementaciones parciales. En una clase abstracta se puede declarar un método abstracto. Éstos son métodos que no contienen implementación, es decir, no está definido el cuerpo del método, sólo su signatura.

Las clases que hereden de una clase abstracta deberán completar la implementación, es decir, deberán implementar todos los métodos abstractos de la clase base (si no lo hacen, serán ellas mismas abstractas).

El objetivo de las implementaciones parciales es dar una interfaz común a todas las clases derivadas de una clase base abstracta, incluso en los casos en los que la clase b ase no tiene la suficiente información como para implementar el método.

Por ejemplo:

```
public abstract class Persona {
  // resto de la clase se mantiene
  public abstract boolean esDocente();
}
```

```
public class Alumno extends Persona {
  // resto de la clase se mantiene
  public boolean esDocente(){
    return true;
  }
}
```

```
public abstract class Empleado extends Persona {
  // resto de la clase se mantiene
}
```

```
public class Profesor extends Empleado {
  // resto de la clase se mantiene
  public boolean esDocente(){
    return true;
  }
}
```

```
public class Bedel extends Empleado {
  // resto de la clase se mantiene
  public boolean esDocente(){
    return false;
  }
}
```

6 Interfaces

Un interfaz es una clase puramente abstracta formada por métodos públicos abstractos.

En Java no se permite herencia múltiple, es decir, sólo se puede heredar de una clase. Pero esto es en ocasiones muy restrictivo. Por ejemplo, si queréis definir una clase applet `MiApplet` tiene heredar de la clase `Applet`. Pero si a la vez queréis que sea sensible a los eventos de ratón, tendría que heredar de la clase adaptador de ratón, pero no puede, ya que sólo existe una línea de herencia.

La “solución” es que se permite que una clase herede de una única clase base, pero pueda implementar varios interfaces. En el caso anterior, `MiApplet` heredará de `Applet` e implementará el interfaz `MouseEventListener`.

Si una clase implementa un interfaz, quiere decir que implementa todos los métodos abstractos de ese interfaz. Esto se representa con la palabra reservada `implements`:

```
class ClaseDerivada extends ClaseBase implements Interfaz1, Interfaz2
{...}
```

Tema 9: Estructuras de almacenamiento: Arrays

Contenidos

1. Declaración de arrays
2. Creación e inicialización de arrays
3. Acceso a los arrays
4. Copia de arrays
5. Uso de arrays

1 Declaración de arrays

Definición:
Un array es una secuencia de objetos o de datos primitivos, todos del mismo tipo, unidos bajo un identificador común. Cada objeto o dato del array está asignado a un posición concreta del array, designada por un índice.

Declaración de arrays:

```
int[] arrayEntero;
int arrayEntero[];

Alumno[] arrayAlumno;
Alumno arrayAlumno;
```

Se indica el tipo de los elementos contenidos por el array, seguido por [] y seguido por el nombre identificador del array (los [] también pueden ir detrás del nombre).

2 Creación e inicialización de arrays

El array es un objeto, que se crea usando new, seguido del nombre del tipo de los elementos del array, seguido por el tamaño del array (entre corchetes). El tamaño del array tiene que indicarse en el momento de su creación, y no puede cambiarse después. El tamaño puede ser un valor constante, o una variable calculada durante la ejecución del programa.

```
arrayEntero = new int[20];
arrayAlumno = new Alumno[3];
```

Igual que sucede con el resto de objetos, se puede declarar el array y crearlo en la misma sentencia:

```
int[] arrayEntero = new int[20];
```

Lo que sucede durante la creación del array es que se reserva memoria para guardar la cantidad de datos indicada, y se asigna la dirección de esa área de memoria a la referencia del array:

- en el caso de arrays de tipos primitivos, se reserva espacio para guardar los valores concretos, ya que el tamaño de los tipos primitivos es conocido a priori.

- en el caso de arrays de objetos, se reserva espacio para guardar las referencias a los objetos. Posteriormente, se crearán los objetos asociados a cada posición del array:

```
arrayAlumno[0] = new Alumno(10038793);
arrayAlumno[1] = new Alumno(10038794); //etc.
```

La representación de la asignación de memoria en la creación de arrays se puede ver a continuación:

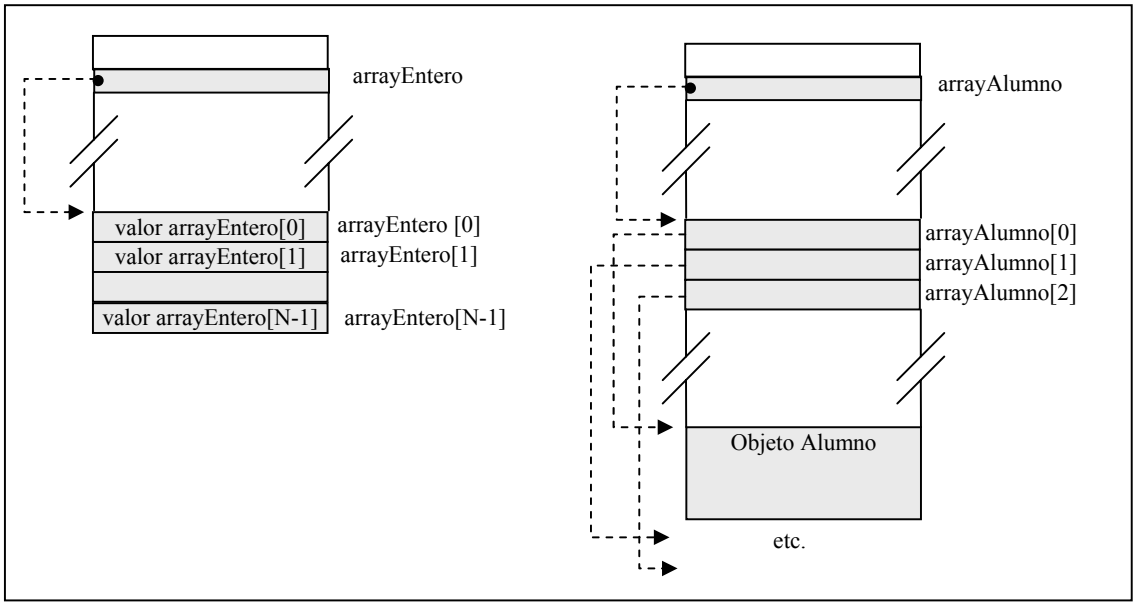

Figura 1: Arrays de valores primitivos y de objetos

Otra forma de inicializar un array es de manera explícita en el momento de su declaración:

```
int[] arrayEntero = {1,2,3,4,5};
Alumno[] arrayAlumno = {new Alumno(10038793),
                        new Alumno(10038793),
                        new Alumno(10038793)};
String[] diasSemana = {"lunes", "martes","miercoles",
                       "jueves","viernes","sabado","domingo"};
```

En los dos últimos casos, son arrays de objetos. Con esta forma de inicializarlos, se crea a la vez el array y los objetos asociados.

3 Acceso a los arrays

Los objetos de tipo array proporcionan dos formas de acceder a su información:

1. **Consulta de su longitud**: Todos los objetos de tipo array tienen un atributo de tipo entero positivo (de sólo lectura) llamado length que proporciona el tamaño del array:

```
nombreArray.length
```

2. **Indexación**: Se puede acceder a todos los elementos del array a través de su índice. El índice de un array Java siempre tiene un rango entre 0 (para el primer elemento

del array) hasta N-1 (para el último), siendo N el tamaño del array.
Para acceder al elemento de índice i se escribe `nombreArray[i]`.

Ejemplo 1:

```
public class Recorrido {
   public static void main (String[] args){
     int tamanyo = Integer.parseInt(args[0]);
     int[] a1 = new int[tamanyo];
     int suma = 0;
     for (int i = 0; i < a1.length; i++) {
        a1[i] = i + 1;
        suma += a1[i];
     }
     System.out.println("La suma es: " + suma);
   }
}
```

Ejemplo 2:

```
public class Arrays {
   public static void main (String[] args){
     int[] a1 = {1,2,3,4,5};
     int[] a2;
     a2 = a1;
     for (int i = 0; i < a2.length; i++)
        a2[i]++;
     for (int i = 0; i < a1.length; i++)
        System.out.println("a1[" + i + "] = " + a1[i]);
   }
}
```

¿Notáis algún comportamiento "extraño en este programa? Efectivamente, la sentencia: `a2 = a1`; no copia el contenido del array, sino sólo la referencia. Es decir, a2 es un alias de a1, y ambas referencian el mismo objeto, por lo que los cambios en los elementos de a2 se reflejarán también en a1.

4 Copia de arrays

Para copiar el contenido de un array en otro, se puede utilizar un bucle:

```
public class EjemploCopia1 {
   public static void main (String[] args){
     int[] a1 = {1,2,3,4,5};
     int[] a2 = new int[a1.length];
     for (int i = 0; i < a1.length; i++){
        a2[i] = a1[i];
        a2[i]++;
     }
     for (int i = 0; i < a1.length; i++)
        System.out.println("a1[" + i + "] = " + a1[i]
                        + "; a2[" + i + "] = " + a2[i]);
```

```
    }
}
```

También se puede hacer directamente mediante el método estático:

```
System.arraycopy(Object src, int src_position, Object dest,
                 int dest_position, int length)
```

Como en el ejemplo adaptado:

```
public class EjemploCopia2 {
   public static void main (String[] args){
     int[] a1 = {1,2,3,4,5};
     int[] a2 = new int[a1.length];
     System.arraycopy(a1,0,a2,0,a1.length);
     for (int i = 0; i < a2.length; i++){
        a2[i]++;
     }
     for (int i = 0; i < a1.length; i++)
        System.out.println("a1[" + i + "] = " + a1[i]
                        + "; a2[" + i + "] = " + a2[i]);

   }
}
```

5 Uso de arrays

Los arrays se usan como cualquier otro objeto Java: pueden ser atributos de un objeto, parámetros de un método (por ejemplo, el main), variables locales o resultados de un método.

Se muestra a continuación un ejemplo de un método que devuelve un array como resultado:

```
public class ArrayFactorial {
   public static int[] generaArrayFactorial(int total){
     int[] factorial = new int[total+1];
     factorial[0] = 1;
     for (int i = 1; i <= total; i++)
        factorial[i] = i * factorial[i-1];
     return factorial;
   }

   public static void main (String[] args){
     int tamanyo = Integer.parseInt(args[0]);
     int[] fact = generaArrayFactorial(tamanyo);
     for (int i = 0; i < fact.length; i++)
        System.out.println("Factorial de " + i
                                  + ": " + fact[i]);
   }
}
```

Tema 10: Búsqueda y ordenación sobre arrays

Contenidos

1. Introducción
2. Inserción ordenada
3. Búsqueda binaria
4. Inserción ordenada binaria
5. Ordenación de arrays

1 Introducción

En este tema se van a introducir una serie de técnicas de acceso eficiente a datos. Para acceder de forma eficiente a los datos, es necesario que éstos estén ordenados. En este tema, vamos a ver distintos algoritmos de ordenación y búsqueda, y su implementación sobre arrays en Java.

Los ejemplos de este tema van a tomar como base la siguiente clase Almacen:

```
public class Almacen {
  private int[] datos;
  private int numDatos;

   public Almacen (int tamaño) throws Exception {
     if (tamaño < 1)
        throw new Exception ("Tamaño insuficiente");
     datos = new int[tamaño];
     numDatos = 0;
  }

   public boolean estaVacio() {
     return numDatos == 0;
  }

   public boolean estaLleno() {
     return numDatos == datos.length;
  }

   public void añadir(int valor) throws Exception {
     if (estaLleno())
        throw new Exception ("Almacén lleno. Imposible añadir");
     datos[numDatos] = valor;
     numDatos++;
  }

   public int buscar (int valor) {
     for (int i = 0; i < numDatos; i++) {
        if (datos[i] == valor)
           return i;
     }
     return -1;
  }

   public boolean eliminar (int valor) {
```

```
        int pos = buscar(valor);
        if (pos < 0)
           return false;
        for (int i = pos; i < numDatos - 1; i++) {
           datos[i] = datos[i+1];
        }
        numDatos--;
        return true;
     }
  }
```

Esta clase representa un almacén de datos de un tamaño máximo dado. El tamaño máximo es la longitud del array de datos, que se establece al llamar al constructor. En el atributo numDatos se guarda el número de datos que se encuentran en el array en un momento dado.

Uno de los usos más frecuentes de los arrays es la ordenación de datos. A lo largo de este tema vamos a ver distintos algoritmos para:

- añadir elementos a un array ordenado
- buscar elementos de un array ordenado
- ordenar un array desordenado

Los distintos algoritmos se diferencias entre sí por su eficiencia, es decir, por el número de operaciones que tienen que realizar para resolver un problema.

Los arrays pueden almacenar un número muy elevado de elementos, por eso es muy importante saber programar algoritmos eficientes.

En este tema se mostrará la implementación de algoritmos de búsqueda y ordenación usando arrays. Sin embargo, los algoritmos se pueden implementar también usando otras estructuras de almacenamiento (Vectores, etc.)

2 Inserción ordenada

El objetivo es, dado un array ordenado (vamos a suponer para nuestros ejemplos ordenación ascendente), insertar un nuevo elemento en su posición correspondiente, de forma que al final todo el array vuelva a quedar ordenado.

Este primer algoritmo que se muestra es el más sencillo y menos eficiente. Consiste en recorrer el array (ya ordenado) comparando los valores de cada elemento ya existente con el nuevo elemento. Una vez se ha encontrado la posición adecuada, se corren todos los elementos de mayor valor una posición a la derecha, para “hacer hueco” al nuevo elemento.

Se puede o bien empezar a comparar desde el principio del array o desde el final. En este ejemplo, se compara desde el principio:

```
  public void añadirOrdenadoInsercion(int valor) throws Exception {
    if (estaLleno())
       throw new Exception ("Almacén lleno. Imposible añadir");
    int i = 0;
    while (i < numDatos && valor > datos[i]) {
       i++;
```

```
      }
      for (int j = numDatos; j > i; j--){
         datos[j] = datos[j-1];
      }
      datos[i] = valor;
      numDatos++;
   }
```

3 Búsqueda binaria

Cuando queremos buscar un elemento con un determinado valor en un array ordenado, no hace falta recorrer todos los otros elementos. Podemos comenzar comparado con el elemento del medio. Si es mayor que el que esgtamos buscando, continuamos la búsqueda por la mitad izquierda. Si es menor, buscaremos en la mitad derecha. Si es igual, ya lo hemos encontrado. Este algoritmo se aplica recursivamente a cada mitad. Este algoritmo de búsqueda tiene una complejidad logarítmica en vez de lineal (el tiempo de ejecución es proporcional al logaritmo en base dos de la longitud del array, en vez de ser proporcional a la longitud del array).

```
   public int buscarBinaria (int valor) {
      int inicio = 0;
      int fin = numDatos - 1;
      int medio;
      while (inicio <= fin) {
         medio = (inicio + fin)/2;
         if (valor > datos[medio])
            inicio = medio + 1;
         else if (valor < datos[medio])
            fin = medio - 1;
         else
            return medio;
      }
      return -1;
   }
```

4 Inserción ordenada directa (binaria)

De la misma forma que hemos aplicado el algoritmo de úsqueda binaria para encontrar un elemento en el array, podemos aplicarlo como parte del método de inserción ordenada. Recordemos que ese método comienza buscando la posición adecuada del elemento dentro del array. Esto lo haremos ahora por búsqueda binaria en vez de por búsqueda lineal. Luego, seguiremos teniendo la parte de “hacer hueco” para el nuevo elemento.

```
   public void añadirOrdenadoInsercionDirecta(int valor)
                     throws Exception {
      if (estaLleno())
         throw new Exception ("Almacén lleno. Imposible añadir");
      int inicio = 0;
      int fin = numDatos - 1;
      int medio = 0;
      while (inicio <= fin) {
         medio = (inicio + fin)/2;
         if (valor > datos[medio])
            inicio = medio + 1;
```

```
            else
               fin = medio - 1;
         }
         int j;
         for(j=numDatos; j>inicio; j--)
            datos[j] = datos[j-1];
         datos[j] = valor;
         numDatos++;
      }
```

5 Ordenación por inserción y por inserción directa

De la misma forma que se aplican los algoritmos de inserción y de inserción directa para añadir un nuevo elemento al array, se pueden aplicar para ordenar un array que estuviera desordenado.

En ambos casos, se va tomando siempre un nuevo elemento del array (empezando por el elemento en en la posición 1) y se añade al subarray de la izquierda, que ya está ordenado.

Se muestra en primer lugar el método de ordenación por inserción de arrays. En este caso, en vez de buscar la posición de forma lineal de izquierda a derecha, se hace de derecha a izquierda, a la vez que se van desplazando los elementos cuyo valor es mayor que el elemento a insertar una posición a la derecha, para ir "haciendo hueco".

```
public void ordenarInsercion() {
  int j;
  for (int i = 1; i < numDatos; i++) {
     int tmp = datos[i];

  _
     for (j = i; (j > 0) && (tmp < datos[j-1]); j--)
        datos[j] = datos[j-1];
     datos[j] = tmp;
  }
}
```

En segundo lugar se muestra el método de ordenación por inserción directa, en el que se aplica el mismo algoritmo de inserción usando búsqueda binaria:

```
public void ordenarInsercionDirecta() {
  int j;
  for (int i = 1; i < numDatos; i++) {
     int tmp = datos[i];

  _
     int inicio = 0;
     int fin = i - 1;
     int medio = 0;
     while (inicio <= fin) {
        medio = (inicio + fin)/2;
        if (tmp > datos[medio])
           inicio = medio + 1;
        else
           fin = medio - 1;
```

```
            }

            for(j = i; j > inicio; j--)
               datos[j] = datos[j-1];
            datos[j] = tmp;
         }
      }
```

6 Ordenación por selección

Finalmente, vamos a ver un mecanismo diferente de ordenación de arrays. Los dos anteriores se han llamado de ordenación por inserción porque han tomado los siguientes elementos a insertar del arrays de forma secuencial (según iban apareciendo), y han realizado la búsqueda de la posición final en el momento de hacer la inserción en el subarray ya ordenado.

En el caso de ordenación por selección, se busca el menor de los elementos que quedan por ordenar, y se inserta directamente al final del array ya ordenado. Es decir, la complicación está en la parte de selección del elemento, y no en la parte de inserción del elemento en el array.

```
 public void ordenarSeleccion(){
   for (int i = 0; i < numDatos - 1; i++){
      int menor = datos[i];
      int j;
      int pos = i;
      for (j = i + 1; j < numDatos; j++) {
         if (datos[j] < menor) {
            menor = datos[j];
            pos = j;
         }
      }
      datos[pos] = datos[i];
      datos[i] = menor;
   }
}
```